AFIT/GCE/ENG/92D-12

AN INTELLIGENT REAL-TIME SYSTEM ARCHITECTURE IMPLEMENTED IN ADA

THESIS

Michael Anthony Whelan
Captain, USAF

AFIT/GCE/ENG/92D-12

Approved for public release; distribution unlimited

AFIT/GCE/ENG/92D-12

AN INTELLIGENT REAL-TIME SYSTEM ARCHITECTURE IMPLEMENTED IN ADA

THESIS

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science in Computer Engineering

Michael Anthony Whelan, A.A.S., B.S.E.E.

Captain, USAF

December, 1992

Preface

The research conducted here was motivated by my experiences as a project engineer for the Pilot's Associate program. During the development phases of the program, most of the effort focused on developing the knowledge bases necessary to build a Pilot's Associate with little attention paid to the environment in which such a system would eventually be deployed. I decided to address the issues of real-time and Ada in this thesis to help smooth future efforts to transition "intelligent systems" into real-time environments. I believe the experience and knowledge gained while I pursued that goal to be invaluable.

I am indebted to a number of faculty at AFIT for the knowledge I gained while attending there. First is my thesis advisor, Major Gregg Gunsch, who taught me the meaning of the word "scope". He also provided valuable feedback that allowed this document to at least be readable. Second is Dr. Gary Lamont whose algorithms class taught me more about "computer science" than all other computer classes combined. Major Eric Christensen assisted me in solving some tricky Ada problems I had given up on as un-solvable. In addition, all my fellow students provided the valuable moral support that allowed me to continue "pressing on".

My most heart felt thanks is reserved for my family. First for my wife, Mary, who ran the household without me and suffered through quite a few "do nothing" weekends. Second are my children, Shaun and Shaunna, who now have a daddy back. Without their understanding and love, I could never have completed this research.

Michael Anthony Whelan

Table of Contents

List of Figures

List of Tables

Abstract

This research begins the process of transitioning real-time intelligent laboratory demonstration programs into the congressionally mandated implementation language Ada. The investigation objective is to analyze the characteristics of real-time intelligent systems and then to design and implement an software architecture capable of supporting the identified characteristics. By beginning to address the specific needs of real-time intelligent systems as implemented in Ada, the path from laboratory demonstration to fielded system is further illuminated.

Conventional real-time systems are fully deterministic allowing for off-line, optimal, task scheduling under all circumstances. Real-time intelligent systems add non-deterministic task execution times and non-deterministic task sets for scheduling purposes. Non-deterministic task sets force intelligent real-time systems to trade-off execution time with solution quality during run-time and perform dynamic task scheduling. Four basic design considerations addressing those tradeoffs have been identified: control reasoning, focus of attention, parallelism, and algorithm efficacy.

Non-real-time intelligent systems contain an environment sensor, a model of the environment, a reasoning process, and a large collection of procedural processes. Real-time intelligent systems add to these a model of the real-time system's behavior, and a real-time task scheduler. In addition, the reasoning process is augmented with a metaplanner to reason about timing issues using the system's behavioral model. Additionally, real-time deadlines force the inclusion of pluralistic solution methods in the intelligent system to allow multiple responses ranging from reactive to fully reasoned and calculated.

AN INTELLIGENT REAL-TIME SYSTEM ARCHITECTURE IMPLEMENTED IN ADA

I. Introduction

> I feel compelled to point out the obvious: a demonstration of some capability (in AI or other technology) on one restricted instance of a general class of problems is important as an existence proof of a technology, but it does not satisfy the general need for a technology that will be able to produce solutions for all unrestricted problems in that class. It is in this sense that I believe that AI will require much basic and engineering research from DoD and other sources for many years to come. Given the utility derived from the relatively modest level of today's technology, I believe that even incremental gains here will prove of phenomenal value to DoD and the economy in general. [Simpson, 1988:1]

> Sec. 8092. Notwithstanding any other provisions of law, after June 1, 1991, where cost effective, all Department of Defense software shall be written in the programming language Ada, in the absence of special exemption by an official designated by the Secretary of Defense [Public Law 101–511].

If you believe the first quotation, then clearly, DoD use of Artificial Intelligence (AI) techniques requires a method to create and field AI systems in Ada. Additionally, the relatively new AI field of Real-Time Intelligent Systems is subject to the same congressionally mandated use of the Ada programming language. Thus Real-Time Intelligent Systems research must begin to focus some attention on implementation issues associated with Ada. The research conducted here attempts to do just that: investigate some implementation issues associated with intelligent real-time systems in Ada. By integrating three broad areas of DoD relevant research; artificial intelligence, real-time systems, and Ada, this research incrementally advances the field of AI and assists the DoD in complying with the congressionally mandated use of the Ada programming language.

1.1. Background

To understand the direction this thesis investigation is taking, it is necessary to examine a number of areas of computer science. These areas include expert systems, blackboard systems, associate systems, and real-time systems. The first requirement, however, is a simple definition of what is meant by the term

'intelligent real-time systems'. For the purposes of this thesis, an intelligent real-time system (IRTS) is defined as a computer control system which can perform some initially specified function and which can

1) adapt its control strategy based upon changes in its operating environment,

2) trade off the quality of a solution against the computational time required to calculate a solution in order to adapt to changes in its operating environment, and

3) guarantee the response times of some set of tasks.

Expert systems are one class of intelligent systems. An expert system tries to perform a given task in a method that is comparable to a human expert. Typically an expert system consists of explicitly represented domain knowledge (in the form of a knowledge base) and an inference engine [Klahr, 1986:28; CLIPS-Ada, 1991] [CLIPSRefMan, 1991]. The knowledge base consists of English-like rules of the form: If A and B then C. The inference engine (sometimes called a monitor) arranges the rules for execution and executes them. The knowledge base and inference engine are separate and distinct entities. By adding or deleting knowledge (rules) in the knowledge base, the expert system's level of expertise can be altered. Changing the knowledge base altogether results in an expert in another domain.

Blackboard systems were developed to allow for multiple cooperating expert systems [Nii, 1989:13-82]. The analogy generally used to describe its operation is a group of experts standing around a blackboard trying to solve a multi-disciplinary problem. Information is placed on the blackboard and each expert responds by providing partial solutions from their particular field of expertise. Eventually, the group may solve the problem that no single expert could.

A blackboard system consists of a global database and a collection of knowledge sources (KS) that act upon the data in the global database. Each of the KS's contributes to the problem solving process by identifying goals, contributing partial solutions, or evaluating partial solutions. Eventually, as each of the KS's responds to changes in the blackboard data, a satisfactory solution is reached. The two types of knowledge sources are domain and control. Domain KS's operate on a specific problem and control KS's help in deciding which domain KS's are appropriate to execute.

Associate Systems are an area in AI that have received significant DoD attention and funding. Associate Systems help operators of planes, helicopters, tanks, and submarines cope with the avalanche of

data available to them and make effective decisions to accomplish their mission [Aldern, 1990] [Lambert, 1990]. One of the most heavily funded systems in this category is the DARPA/USAF Pilot's Associate. (Others include Day/Night Adverse Weather Pilotage System, Submarine Operator's Associate System, Rotocraft Pilot's Associate, and a Special Operations Forces spin-off effort from the Pilot's Associate program). To date, each of these systems has used a collection of rule-based approaches and blackboard systems. Each program has also developed a methodology to acquire the appropriate domain knowledge and represent that knowledge in a form the system implementor can use [Aldern, 1991:4-1 to 4-20]. All of these systems so far are proof-of-concept systems. One key concern is the requirement to operate in "real-time". Although research is moving in this direction [Aldern, 1991] [Lambert, 1990] [Dodhiawala, 1988] [Payton, 1991], none currently guarantee operation in real-time and none are implemented in Ada.

Real-time systems usually mean fast systems that operate on temporally valid data. Both the term *fast* and *real-time* are problem specific and precise definitions are only relevant for the specified domain. For example, a real-time system used to monitor continental drift may only be required to take and record measurement samples once a month. On the other end of the spectrum, a space shuttle flight control system may require that hundreds of sensors be checked in milliseconds. The point is that real-time systems must operate fast enough for the particular application. Generally, this requirement is stated by saying that the system's response must be calculated by the required deadline, with little said about how to determine what is the deadline. Thus a intelligent real-time system must exhibit appropriate behavior fast enough in the chosen domain.

1.2. Problem Statement

Due to funding limitations and program constraints, the concept of dynamically controlling the real-time performance of an intelligent system was never fully developed in the Pilot's Associate program. Additionally, both contracted development teams used implementation languages other than Ada. Thus, to date no large intelligent real-time system has been developed and implemented in Ada, and no analysis of such a system performed. This thesis effort proposes to investigate issues in the development of such a system and analyze some of the performance issues raised. By providing a potential design solution, a feasibility demonstration, and some analysis of its performance, this thesis can have a direct impact on future real-time, embedded, intelligent DoD systems.

1.3. Assumptions

This thesis investigation assumes a basic system functional description as outlined in the Pilot's Associate program [Lambert, 1991] [Lambert, 1990] [Aldern, 1990] [Aldern, 1991]. Thus it is assumed that there exists a large collection of known procedures or tasks that are required to perform a given function. In addition, the relationship between these tasks and their impact on the current problems facing the system can be determined. This mapping of functions and relationships to the current context in which the system is operating is referred to as a plan/goal graph or task network [Wilensky, 1983] [Smith, 1990]. It is not the goal of this thesis effort to design from scratch an intelligent real-time system. Rather, this thesis effort investigates the mapping of previous designs to Ada, the addition of components necessary to overcome deficiencies in the previous designs, and the issues arising out of that mapping and addition of components.

No effort is made to acquire the knowledge necessary to add 'intelligence' to the system. Instead, a baseline system similar to the Pilot's Associate is assumed. The goal of this system is not the development of the knowledge base for such a system, rather it is the mapping of the designed system to Ada and to investigate the control of such a system. The application of the architecture resulting from this work to a specific domain is left as future work.

1.4. Scope

This research proposes to examine intelligent real-time systems but is limited primarily to knowledge based systems acting as the intelligent agents in the system. It is not the intent of this research to examine the real-time issues associated with other AI software disciplines, nor are hardware issues addressed. Thus, no effort is being made to consider machine learning, neural networks, genetic algorithms, or any of the other AI disciplines. Similarly, no effort is made to examine hardware architectures, chip designs, or memory systems. It is fully expected that areas requiring performance improvements will be encountered and that these other AI areas and hardware may provide solutions to those problem areas. It is left as future research to incorporate those AI technologies and hardware into the framework this thesis is proposing.

1.5. Approach

The approach used in this thesis effort is typical of most research efforts. First, a literature review of real-time and intelligent real-time systems is conducted. The purpose of the review is to educate myself

and the reader about the issues affecting both traditional real-time systems and intelligent real-time systems. Additionally, the literature review exposes problems with current systems and examines some potentially useful methods of dealing with those problems. Specifically, scheduling real-time tasks using the rate monotonic scheduling theory and methods of scheduling imprecise computations.

From that review, conclusions are drawn about what constitutes an intelligent real-time system. Once the constituent parts are identified, an architecture is presented that incorporates each of those parts. The component most lacking in current intelligent real-time systems is identified and methods of including it into an intelligent real-time system architecture is examined.

Next, a feasibility demonstration system is constructed that investigates incorporating a dynamic real-time task manager into an intelligent real-time system. The feasibility demonstration is constructed to allow for validation of the concepts developed in this thesis and provide insight into potential problem areas. It is not offered as an optimal solution tuned for maximum performance.

Next, an analysis of the implemented system's performance is made. Specifically, an analysis of the dynamic task creation and control strategies is made and the impact of each parameter on system performance identified and quantified in terms of execution speed, code size, and appropriateness for the context. Additionally, performance problems are traced to either a flaw in the implementation or perhaps a more serious flaw in the design.

Finally, an attempt to point the direction of future work in this area is made. Drawing on the results of the efforts of this thesis, a suggested path of continuing research and development is laid out.

1.6. Summary

This research has four objectives. Listed in order of importance, they are

1) development of an integrated set of Ada data and control structures that allows for the implementation of an intelligent real-time system and architecture to support such a system,

2) implementation of the developed data and control structures into an intelligent real-time system and a feasibility demonstration of the developed architecture,

3) an evaluation of the implemented intelligent system architecture to determine the utility of the design and the overhead necessary to support its run-time computational needs, and

4) identification and evaluation of some performance metrics useful in evaluating intelligent real-time systems performance.

1.7. Thesis Overview

This thesis follows the pattern used by most scientific research reports. In Chapter 2, the results of a literature search are presented. Current knowledge in AI systems, real-time systems, and intelligent control is examined and issues important to this thesis are identified. Additionally, the common components of intelligent real-time systems are listed. Chapter 3 addresses some of the possible design approaches for developing an intelligent real-time system and enumerates the design assumptions. The key design decisions made are discussed. Chapter 4 presents the intelligent real-time system architecture vision developed and addresses some of the features that are missing in previous architectures. Chapter 5 provides a detailed look at the developed feasibility demonstration system. The goal of the feasibility demonstration is to provide confidence in the potential of the architecture presented in Chapter 4 and allow for concept testing. Chapter 6 presents the results obtained by testing the feasibility demonstration system and the impact of those results on the proposed intelligent real-time system architecture. Chapter 7 culminates the thesis with conclusions and recommendations.

II. Background

The technology needed to build an intelligent real-time system spans a multitude of engineering disciplines. Those that are relevant to this thesis include traditional real-time system design, real-time scheduling algorithms, real-time system design in Ada, knowledge representation, expert systems, associate systems, blackboard systems, planning, and intelligent control. In order to lay the foundation for understanding the work presented in this thesis, it is necessary to examine some of the more important aspects of these related technologies. The following background information covers traditional real-time systems rate monotonic scheduling, then imprecise computation scheduling, followed by intelligent real-time systems, and culminating with a description of the components needed to implement intelligent real-time systems.

2.1. Traditional Real-Time Systems

Sprunt, Sha, and Lehoczky provide a good introduction to real-time system design issues and algorithms [Sprunt, 1989] [Sprunt, 1990]. They classify tasks based upon the task's *deadline* and *arrival pattern*. Classifications for deadlines are *hard* and *soft*. Hard-deadline tasks are defined as "If meeting a given task's deadline is critical to the system's operation, then the task's deadline is considered to be *hard*" [Sprunt, 1990:2]. Soft-deadline tasks are tasks whose deadline is desirable but not absolutely essential for correct system operation. Additionally, Sprunt and Sha add the category *background task* for those tasks without a timing constraint.

Classifications for task arrival rates include periodic and aperiodic. A periodic task is defined by Sprunt and Sha as a task that arrives at regular, predictable times. Sprunt and Sha include things like sensor updates or monitoring tasks in this category. Aperiodic tasks are defined to be tasks with irregular arrival times and result from "the processing requirements of events with nondeterministic request patterns, such as operator requests" [Sprunt, 1990:2]. Using deadlines and arrival patterns, they essentially divide the types of real-time tasks into four categories: hard-deadline periodic tasks, soft-deadline aperiodic tasks, sporadic tasks, and background tasks. Sprunt and Sha define each task type as follows:

- *Hard-Deadline Periodic Task.* A periodic task consists of a sequence of requests arriving at regular intervals. A periodic task's deadline coincides with the end of its period.

- *Soft-Deadline Aperiodic Task.* An aperiodic task consists of a stream of requests arriving at irregular intervals. Soft deadline aperiodic tasks typically require a fast average response time.

- *Sporadic Tasks.* A sporadic task is an aperiodic task with a hard deadline and a minimum inter-arrival time (the amount of time between two requests).

- *Background Tasks.* A background task has no timing requirements and no particular arrival pattern. Background tasks are typically assigned the lowest priority in the system ... [Sprunt, 1990:2]

Systems with hard-deadline periodic tasks can be constructed efficiently using the *rate-monotonic* scheduling algorithm [Sha, 1991] [Sha, 1989] [Sprunt, 1989] [Liu, 1973]. This algorithm "assigns priorities to tasks as a monotonic function of the rate of a (periodic) function" and assumes a priority driven, preemptive scheduling discipline [Sha, 1991:3]. The rate-monotonic algorithm is the provably optimal scheduling algorithm for preemptive scheduling of hard-deadline periodic tasks [Sprunt, 1990:2]. The rate monotonic theory provides a simple inequality to determine whether a given set of periodic tasks is schedulable. Theorem 1 below provides a sufficiency test that ensures all tasks meet their deadlines. For task sets whose utilization, $U(n)$, exceeds that of (2.1), Theorem 2 below is both a necessary and sufficient test.

Theorem 1: A set of n independent periodic tasks scheduled by the rate monotonic algorithm will always meet its deadlines, for all task phasings, if

$$\frac{C_1}{T_1} + \ldots + \frac{C_n}{T_n} \leq n(2^{1/n} - 1) = U(n) \tag{2.1}$$

where C_i and T_i are the execution times and period of task τ_i respectively and $U(n)$ is the utilization of the task set [Sha, 1989:5].

Theorem 2: A set of n independent periodic tasks scheduled by the rate-monotonic algorithm will always meet its deadlines, for all task phasings, if and only if

$$\forall i,\ 1 \leq i \leq n, \quad \min_{(k,l) \in R_i} \sum_{j=1}^{i} C_j \frac{1}{lT_k} \left\lceil \frac{lT_k}{T_j} \right\rceil \leq 1 \tag{2.2}$$

where C_i and T_i are the execution times and period of task τ_i respectively and $R_i = \{(k, l) \mid 1 \leq k \leq i,\ l = 1, \ldots, \lfloor T_i / T_k \rfloor\}$. i represents the task to be checked, k represents the tasks of equal or higher priority, and l represents the scheduling points for task i [Sha, 1989:5].

In Theorem 1, for large values of n, $U(n)$ converges to 0.69 ($ln\ 2$). Assuming the worst case scenario where all tasks are started simultaneously, Theorem 2 checks if each task can complete its execution by its first

deadlines at each *scheduling point* before task i's deadline. A scheduling point for a task τ is the deadline of each task with a deadline before τ's.

In addition to using a priority driven, preemptive scheduling discipline, Theorems 1 and 2 also assume, process switching is instantaneous, the tasks account for all the execution (i.e. the operating system consumes zero time), the tasks do not interact, tasks become ready to execute exactly at the beginning of their periods and, task deadlines are always at the end of the period. Continuing research is addressing some of the limitations associated with the assumptions used to develop Theorem 1 and Theorem 2 [Sha, 1991] [Klein, 1990]. First, the effect of non-zero task switching times is addressed by modifying the execution time of a task, C_i to include context switching times, C_S. In Theorems 1 and 2, the new task execution time becomes $C'_i = C_i + 2C_S$ since the processing context is switched at the beginning and end of each task's execution.

Task synchronization was addressed by the development of the *priority ceiling protocol*. "The priority ceiling protocol has two important properties, 1) freedom from mutual deadlock and 2) bounded priority inversion, which means at most one lower priority task can block a higher priority task during each task period" [Sha, 1991:6]. Inclusion of the priority ceiling protocol allowed the formulation of the two additional theorems:

> *Theorem 3*: A set of n periodic tasks using the priority ceiling protocol can be scheduled by the rate monotonic algorithm, for all task phasings, if the following condition is satisfied [Sha, 1989:15].
>
> $$\frac{C_1}{T_1} + \ldots + \frac{C_n}{T_n} + \max\left(\frac{B_1}{T_1}, \ldots, \frac{B_{n-1}}{T_{n-1}}\right) \leq n(2^{\frac{1}{n}} - 1) \tag{2.3}$$
>
> *Theorem 4*: A set of n periodic tasks using the priority ceiling protocol can be scheduled by the rate monotonic algorithm, for all task phasings, if and only if the following condition is satisfied :
>
> $$\forall i,\ 1 \leq i \leq n, \quad \min_{(k,l)\in} R_i\left(\sum_{j=1}^{i-1} C_j \frac{1}{lT_k}\left\lceil \frac{lT_k}{T_j}\right\rceil + \frac{C_i}{lT_k} + \frac{B_i}{lT_k}\right) \leq 1 \tag{2.4}$$
>
> where C_i, T_i, and R_i are defined in Theorem 2, and B_i is the worst-case blocking time for task i [Sha, 1989:15].

Theorems 3 and 4 provide generalized forms of Theorems 1 and 2 that handle blocking due resource sharing or task synchronization. To handle soft-deadline aperiodic tasks the Deferrable Server and Priority Exchange algorithms can be used [Sprunt, 1989:11]. Both of these approaches create a high priority

periodic task (called a server) for servicing aperiodic tasks. The goal of each algorithm is to preserve the resource bandwidth allocated to aperiodic tasks; allowing immediate execution of the aperiodic task when it arrives. In the case of the Deferrable Server, during each server period, the aperiodic task's time is held for the entire period and replenished at the end of the server's period. Thus when an aperiodic task arrives, it is serviced immediately if time remains in the aperiodic task's time budget.

The Priority Exchange algorithm, as the name implies, exchanges its priority with lower priority periodic tasks whenever there are no aperiodic tasks ready for execution. At the end of the server's period, the aperiodic tasks priority is again raised to the highest level. Again, since the aperiodic task always has the highest priority, the Priority Exchange algorithm ensures that aperiodic tasks are handled immediately when they arrive if time remains in the aperiodic task's time budget. The difference between the two algorithms lies in their complexity and schedulability bounds as discussed by Sprunt, Sha, and Lehoczky [Sprunt, 1989:6-10].

The point of this discussion is that mathematically precise methods exist for real-time system design of systems composed of static priority, hard-deadline, periodic, sporadic, and/or aperiodic tasks. The importance of rate-monotonic scheduling theory for use in intelligent real-time systems lies in its feasibility test and corresponding priority assignment. The feasibility of a periodic task set can be quickly calculated and the result acted upon. Additionally, the scheduling problem associated with assigning priorities is significantly reduced if not eliminated.

2.2. Scheduling Imprecise Computations

The preceding discussion dealt with methods to determine the feasibility of a task schedule and schedule fairly well defined tasks to achieve real-time performance. There exists another class of tasks that vary in processing times and so cannot be handled easily by the preceding methods. The imprecise computation technique can be used to help schedule tasks that fall into this category [Liu, 1991]. It allows tradeoffs between result quality and computation time. This quality/time trade-off helps prevent timing faults and assists in achieving graceful degradation.

Liu, Lin, Shih and Yu explain that their basic strategy revolves around the division of all time critical tasks into two subtasks: a mandatory subtask that provides an adequate result and an optional subtask that provides a refined result [Liu, 1991:58]. If the optional subtask is scheduled and executes to completion,

then the result is said to be *precise*. If the optional task is terminated before it completes, and the mandatory task has been completed, then the result is said to be *imprecise*.

Three methods and the associated costs of each method for creating mandatory and optional subtasks are discussed by them [Liu, 1991:59]. The first method is the *milestone method* and takes advantage of *monotone* tasks. A monotone task is one whose result quality does not decrease as the task executes. If the results of the task's execution and error indicators are recorded at appropriate instances, then the task's result grows more precise as it is allotted more time to execute. The user can then decide, based upon the error indications and result, when to terminate the task. The cost of the milestone method is the storing of the intermediate results.

The second method described by them is referred to as a *sieve function* [Liu, 1991:59]. In this method, computation steps (and therefore result quality) are traded off for processing time. The example given to illuminate how this method is employed involves using a previous cycle's noise level estimate when examining this cycle's radar returns. The cost for sieve functions is higher scheduler overhead. Liu, Lin, Shih and Yu classify this type of scheduling problem as a *0/1 constraint problem* [Cormen, 1990:335]. Since no benefit is gained unless the entire sieve function completes before its deadline, it should either be scheduled to execute to completion or not scheduled at all.

The third method they describe is the *multiple version* method [Liu, 1991:59]. As the name implies, each task has two versions; a primary version and an alternate version. The primary version has a longer execution time, but produces a precise result. The alternate version produces a less precise result in a shorter time. The cost of the multiple version method includes both storage space for the multiple versions and higher scheduler overhead. Again, Liu, Lin, Shih and Yu classify the multiple version method as a 0/1 constraint problem for the same reasons cited above. In this case, scheduling the primary version is considered to be the same as scheduling both the mandatory subtask and the optional subtask. Scheduling the alternate version corresponds to scheduling only the mandatory subtask.

Liu, Lin, Shih and Yu then go on to develop a basic workload model for all imprecise computation methods they described [Liu, 1991:59-60]. Given a set of preemptable tasks,

$$T = \{T_1, T_2, \ldots, T_n\}$$

the tasks T_i in the set can be described by the following parameters:

- r'_i Ready time at which time T_i becomes ready for execution
- d'_i Deadline by which T_i must be completed
- τ_i Processing time required to execute T_i to completion
- w_i Weight that measures the relative importance of T_i
- M_i Mandatory subtask of T_i
- O_i Optional subtask of T_i
- m_i Processing time required to execute M_i to completion
- o_i Processing time required to execute O_i to completion

Note that in the preceding definitions, $m_i + o_i = \tau_i$. Additionally, the deadlines of the subtasks M_i and O_i are the same as the deadline for T_i. The ready time for the M_i subtask is the same as T_i but the O_i does not become ready until after the mandatory subtask completes.

A valid schedule of T is defined as one that "assigns the processor to at most one task at any time, and every task is scheduled after its ready time. Moreover, the total length of the intervals in which the processor is assigned to T_i, ..., is at least equal to m_i and at most equal to τ_i." [Liu, 1991:59]. Additionally, a valid schedule is termed *feasible* if every task is completed by its deadline. Schedulable task sets have at least one feasible schedule.

One final parameter used by Liu, Lin, Shih and Yu is the error parameter, ε_i. This parameter is used to calculate a value for processing the optional subtasks. The error parameter is calculated by the equation $\varepsilon_i = E_i(o_i - \sigma_i)$. Here, σ_i is the amount of processor time allotted to O_i, and E_i is assumed to be a monotone non-increasing function of σ_i.

Once the imprecise computation scheduling problem is defined in these terms, it becomes a matter of applying an appropriate algorithm to obtain a particular scheduling goal. Scheduling goals can be designed to minimize the total error, average error, the number of discarded optional tasks, the number of tardy tasks, or the average response time [Coffman, 1976] [Liu, 1991]. It becomes the job of the system designer or the run-time controller to determine the appropriate scheduling goal at any particular instant during the system's operation.

2.3. *Intelligent Real-Time Systems*

As vague as it may sound, real-time in an intelligent system seems to mean "fast enough". Traditionally, as discussed above, a real-time system's performance is dictated by hard limits imposed on the amount of time a task has to execute. These limits are normally pre-defined by the process that is being controlled and the speed of the equipment used to control that process. For example, in a flight control system the real-time requirements of the digital computer are dictated by the speed of the control mechanisms which move the control surfaces, the speed with which the instruments monitoring the aircraft attitude can detect changes resulting from control surface movements, and the time that the laws of aerodynamics establish a response must be made. In an intelligent real-time system, real-time performance is context dependent. The response time required of an autonomous aircraft from the detection of a threat depends upon how quickly a response is required to ensure a successful outcome. The system obviously must respond much faster to an inbound missile targeted at the ownship than it does to a threat aircraft which is beyond the lethal radius of any weapons it can carry.

The key point of this type of variable response timeline is that it places the additional burden upon the system of being able to determine the time constraints at any particular instant. A run-time, dynamically adaptable control structure appears to be required [Payton, 1991] [Lambert, 1990] [Lizza, 1989] [Stankovic, 1988]. The system must be able to respond to rapidly changing requirements and still produce valuable output. Two DoD funded programs, Pilot's Associate and Adaptive Suspension Vehicle, have been addressing these issues. The programs, as they relate to intelligent real-time systems, are examined below.

2.3.1. Pilot's Associate Program. Basically, the Pilot's Associate program is developing an electronic back-seater to assist tomorrow's single-seat fighter pilot cope with the ever increasing amounts of information presented to him [Whelan, 1990] [Banks, 1991] [Aldern, 1990] [Lambert, 1990] [Aldern, 1991]. The Pilot's Associate is developing a system to help the pilot identify the crucial information from the background noise. It is the system which makes the aircraft understand the pilot's objectives, preferences, and restrictions, and works in the tireless fashion of the computer in complying with them.

Figure 2.1 shows the overall concept of a Pilot's Associate. The system consists of six functional elements: Situation Assessment (SA), Mission Planner (MP), Tactics Planner (TP), Pilot-Vehicle Interface (PVI), System Status (SS), and the System Executive (SE). SA is responsible for assessing the world

external to the aircraft. MP is responsible for global planning from take-off to landing. TP is responsible for responding to immediate threats. PVI ensures the system provides what the pilot wants, when he needs it. SS is responsible for assessing the world internal to the aircraft. And finally, SE is responsible for ensuring that all elements are solving not only the same problem but also the right problem. In the following discussions, the term *system* refers to the entire collection of functions that make up the Pilot's Associate. A *sub-system* or *module* is one particular functional element (e.g. Tactics Planner, Pilot Vehicle Interface). A *machine* is a computer that pieces of the functional elements are implemented upon. (Note that originally, the system was developed with each module to reside upon a different machine. However, as the system developed over time, it became apparent that different pieces of the modules needed to closely work together. This forced a migration of module pieces together based primarily upon data usage).

2.3.2. Lockheed's Pilot's Associate The current Lockheed version of the Pilot's Associate is probably the furthest along in trying to implement a fairly complex intelligent real-time system [Aldern, 1991]. Essentially, the approach is to code the system in C++ and then to statically distribute the set of coded tasks across as many processors as necessary to achieve real-time performance. The process is based upon the premise that processing power will be available (or can be added) to operate these systems in real-time and *thus no separate executive for controlling execution exists*. This approach has some potential problems.

First, the domain chosen for these systems, by their very nature, is dynamic. It is widely held that predictability in any combat situation will lead to higher risk of death or failure. Thus systems that operate in a combat situation, or any competitive situation, face continually changing environments as one system tries to overcome or defeat another. Any system designed for a competitive environment must be able to change quickly as new knowledge is obtained. In the current Lockheed Pilot's Associate, since the knowledge acquired is translated directly into an implementation language (C++), changes potentially require significant work to locate the piece of code to change, determine how to change it, code the changes, and tune the system performance to again ensure real-time constraints can be meet [Aldern, 1991:4-6 to 4-20]. Essentially, the performance goals of the program have eliminated what was the knowledge base as a separate entity and distributed the knowledge throughout the system. Given the syntax of the Ada language, this same 'knowledge coding' approach may be necessary for any DoD system.

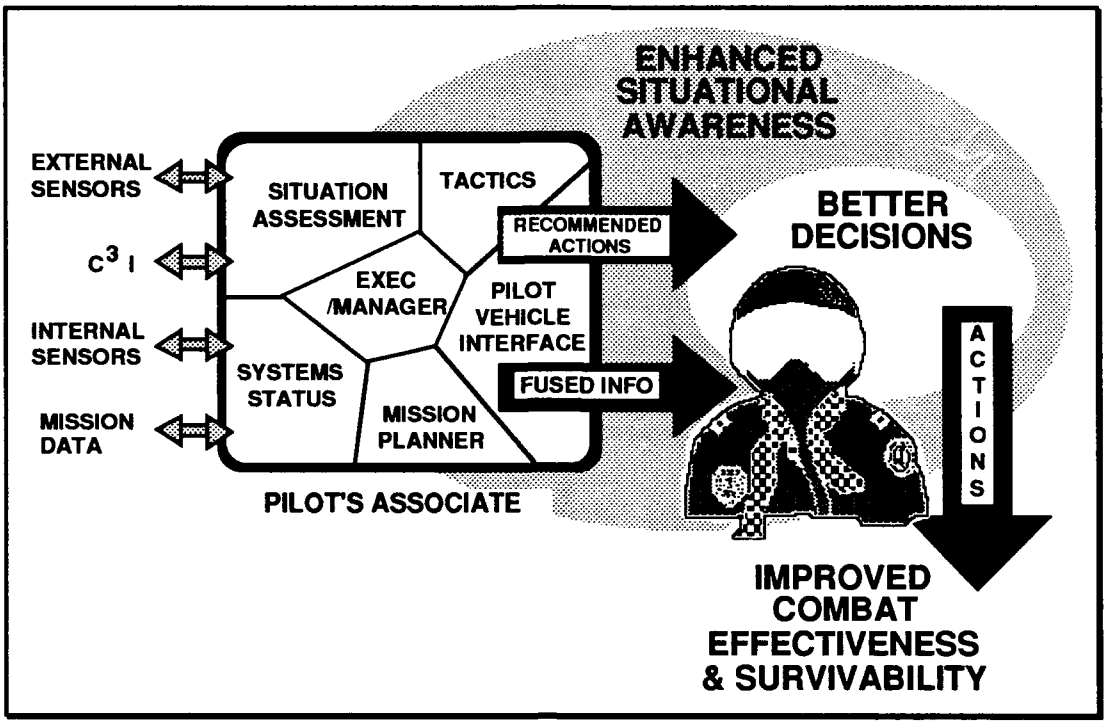

Figure 2.1 Overall Concept of the Pilot's Associate

Additionally, the computational times for a large percentage of AI approaches are generally non-deterministic (i.e., the search space is so large that an exhaustive search is impossible, thus some sort of guiding heuristics are used to get a solution that satisfies some domain specific criteria). Traditional real-time systems do not handle non-deterministic tasks well and mapping an intelligent system's approach into a traditional real-time system usually means modifying the original problem to eliminate some of the difficulties associated with real-time operation. Generally, tasks are off loaded from the computer to the human operator. If no human operator exists, as is the case in a fully autonomous system, the system generally is given a 'canned' response to apply in all conditions.

Finally, this approach assumes there always is, or will be able to obtain the necessary processing power to handle any possible workload. Although this may be true, I tend to believe that the opposite is true; namely that there will never exist enough processing power to solve all problems simultaneously. This should not be taken as a condemnation of the current Pilot's Associate approach. For example, one can envision the problem as a standard set partitioning problem where the sets are combinations of the tasks and the goal is to partition those tasks among *n* processors as efficiently as possible [Brassard, 1988:92] [Cormen, 1990:946] [Tindell, 1992]. If the initial engineering design effort has ensured that some amount

of excess computing power exists, the problem is then reduced to creating a tool to solve the set partitioning problem. At the point where no more additional tasks can be added, one simply upgrades the existing processors to newer and faster versions, or adds another.

The Lockheed Pilot's Associate program is currently not experiencing any major run-time problems with this method. However, the system is not completed and is known to have reduced functionality because of the real-time constraints. If at some future point, processing power becomes an insurmountable problem, a major system redesign will be required. Because of this constraint, Lockheed's real-time method will not be used in this thesis investigation.

2.3.3. McDonnell Aircraft's Pilot's Associate McDonnell Aircraft Company's (McAir) Pilot's Associate program real-time efforts were divided into two major thrust areas. McAir tackled the actual implementation and sub-contracted FMC Corporation to investigate the software design and architecture issues. Since McAir's implementation built upon the work FMC did, a review of the general model FMC created is done first. Following that discussion, the McAir implementation of that model is examined. The discussion is fairly in-depth since the work by FMC and McAir shapes the direction of this thesis effort.

Intelligent real-time system analysis by FMC [Dodhiawala, 1988] reached two conclusions:

1. In a system such as the Pilot's Associate, there will never be enough processing power to do all required tasks, which dictates then,

2. Real-time performance cannot be obtained by speed alone.

Although the number and speed of the processors used in implementing a real-time system are a primary concern, they are not by any means the only concerns in a complex intelligent real-time system. The software architecture which controls the operation of the system plays an equally large part in achieving real-time performance. FMC examined those characteristics (other than processor type and processor speed) of a complex, real-time, intelligent system that affect it's real-time performance.

FMC's research determined there are four basic dimensions of an intelligent real-time system; *speed*, *responsiveness*, *timeliness*, and *graceful adaptation* [Dodhiawala, 1988:1-2]. Speed is the dimension concerned with the number of tasks executed per time unit. Speed can be increased by more and faster processors or by increasing the efficiency of the algorithms. Responsiveness is the ability of the system to

take on new tasks quickly. It requires that the system be predictive enough to begin composing and executing responses to new developments rapidly, and to appear as though all requests are handled instantly. Timeliness is the system's ability to conform to task priorities. In other words, the system must be able to perform those functions that are the most urgent when they are required. Any responses must be able to be determined, presented, and acted upon in order to affect the current situation, not the past. Graceful adaptation refers to the system's ability to reset priorities based upon the changing processing load of the system. When the workload exceeds the capacity of the system, it must be able to focus its resources on those tasks that are the most crucial for that moment. Clearly, three of these architectural features are dynamic in nature and require run-time control to ensure the desired system performance is obtained.

Starting with the assumption of a blackboard implementation of the intelligent system, FMC believes the run-time adjustable parameters of *execution margin, scheduler heuristics,* and *channel priorities* can be used to form the basis of run-time, closed-loop, control of knowledge-based systems [Dodhiawala, 1988:7]. The execution margin is defined as the number of knowledge sources that are allowed to execute during each pass through the top level control cycle. Scheduler heuristics refers to the rules of thumb used to order the currently active knowledge sources for execution. Channel priorities are run-time assigned values that dictate how important a particular event is. Figure 2.2 shows the internal operation of the reasoning process in an intelligent system that allows adjustment of these parameters. It should be pointed out here that although at the top level, the intelligent system operates in an asynchronous fashion; inside the reasoning process, synchronous control is exercised.

In Figure 2.2, the four channels (represented by the thick gray horizontal lines) are EMERGENCY for processing events immediately, HIGH for processing fairly critical events, AVERAGE for processing routine events such as monitoring the mission plan, and LOW which processes items only when the other three channels are empty. The channel priority assigned to an event determines the relative importance of that event.

The four gray rectangles in Figure 2.2, Trigger, Pre-Condition, Schedule, and Execute, represent the stages a knowledge source goes through. The Trigger step attempts to trigger knowledge sources based on the attributes of either the asynchronous external events or the synchronous internal events. It is important to realize that an event can trigger more than one knowledge source. When a knowledge source is triggered, its variables are set, establishing the context for the execution of the knowledge source or instantiation (called a Knowledge Source Activation Record or KSAR). The PRECONDITION step

establishes whether a valid context exists for the execution of that KSAR. Possible values returned in this step are OBVIATE – the KSAR is irrelevant or inappropriate for the triggering event, NIL – the context has yet to be established, or the preconditions are satisfied and the KSAR moves to the scheduler to be scheduled. In the Schedule step, KSARs are ranked and scheduled for execution based upon the current scheduler heuristics. These heuristics evaluate the KSAR based upon its importance and urgency. An important KSAR is one that is relevant to the current system goal, while an urgent KSAR is one whose deadline for execution is approaching. The Execute step will execute the KSARs based upon the execution margin. The execution margin determines how many KSARs on each priority channel will be allowed to execute during each run through the top level control cycle. Using this approach, and assuming that not all tasks can be processed in real-time, FMC believes that the control overhead involved is justifiable in terms of the total system performance [Lambert, 1990] [Dodhiawala, 1988].

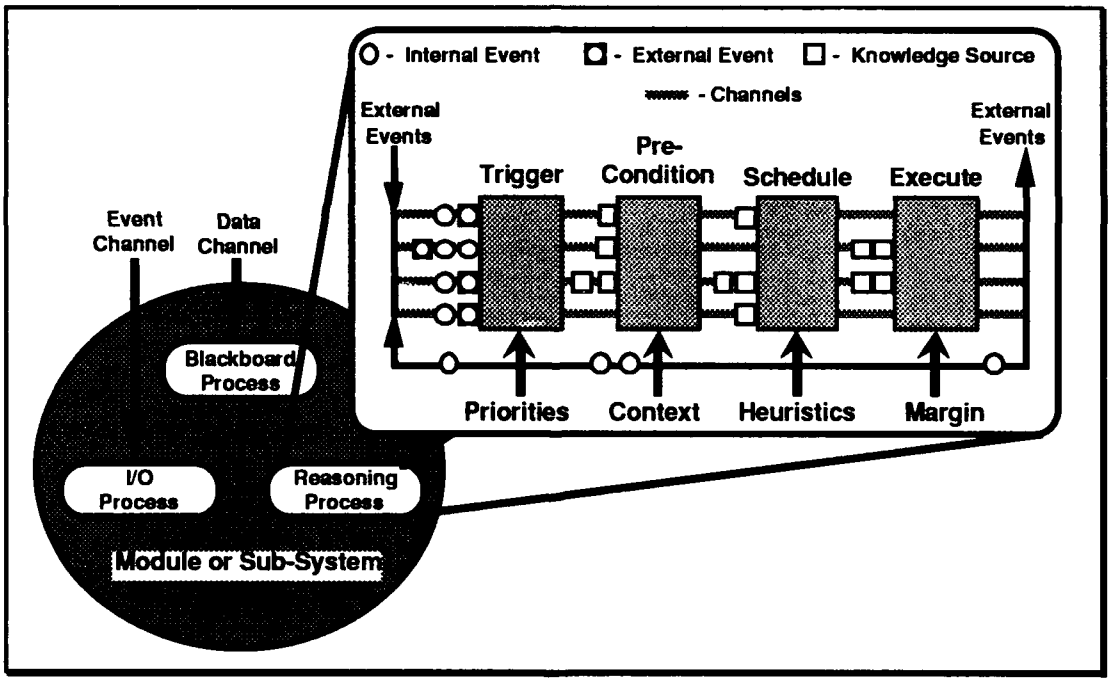

Figure 2.2. Reasoning Process Diagram

McAir's system ran on four Texas Instruments Explorer II machines with an assortment of other machines providing support functions [Lambert, 1991]. The top-level architecture is shown in Figure 2.3. Each of the sub-systems (indicated by the circles) operated asynchronously and ran in parallel on independent machines. The system was event-driven, allowing communication of changes to blackboard

data instead of transmitting entire updated blackboard data structures. The blackboard itself was distributed among each of the machines that make up the system. Each sub-system was further divided into a blackboard process, I/O process, and a reasoning process as shown in Figure 2.2.

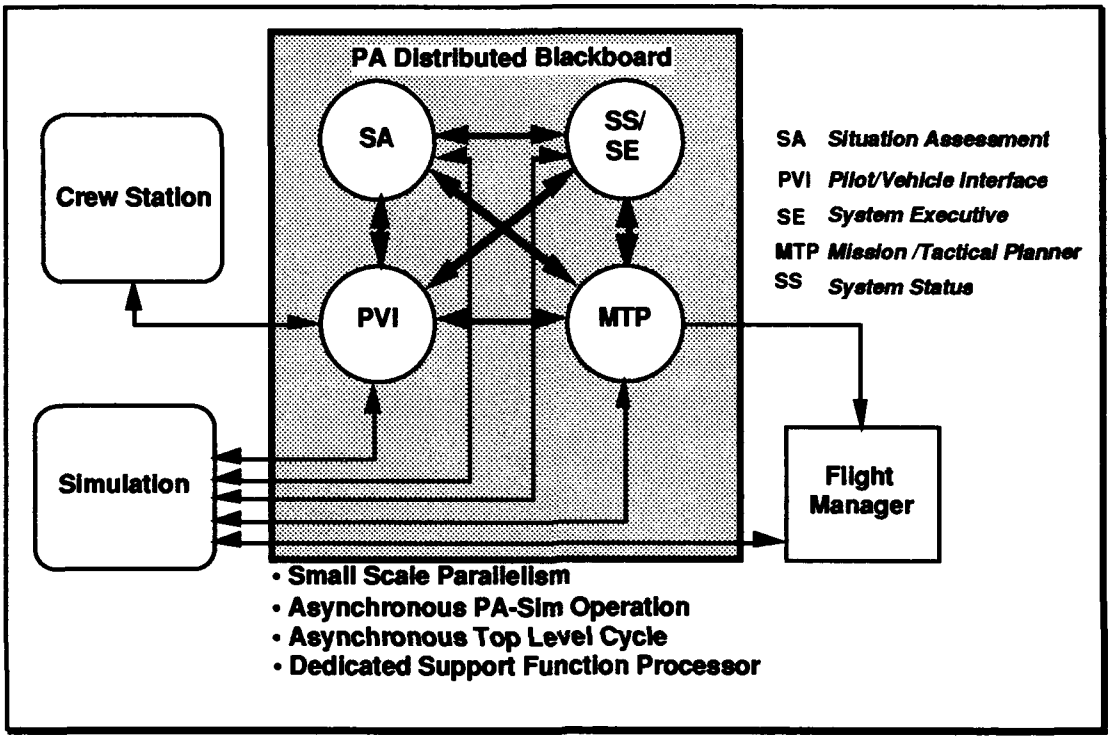

Figure 2.3 McAir's Top Level Architecture

Using FMC's approach, McAir's team incorporated some of the real-time concepts into their system. The system was implemented primarily in Inference Corporation's Automated Reasoning Tool (ART), modified to allow for some PA-specific functions. A modification to the ART agenda mechanism allowed for heuristic-scheduling, a feature from FMC's real-time work. Additionally, the Knowledge Source syntax was modified to put trigger and pre-condition patterns into a context slot and extra declarations were added to include specification of the knowledge source urgency and importance [Lambert, 1991].

The module internal architecture shown in Figure 2.4 implemented the blackboard, I/O, and reasoning processes [Lambert, 1991]. Basically, the ART working memory used Rete Net pattern matching provided with ART to match the context of the knowledge sources [Lambert, 1991] [Fanning, 1990]. Once a knowledge source was matched, it was placed on the agenda for execution. From the agenda, the

knowledge source was executed using a LISP macro that expanded into ART rules. The LISP macro allowed McAir to augment the ART rule representation and include the real-time scheduling features of importance and urgency. The gray lines indicate the control cycle. Again, ART was modified to allow for communication processing to take place after each knowledge source was activated. The control cycle was designed so that the Blackboard and I/O processes slept if not required, thus allowing more of the machines resources to be used in knowledge source execution.

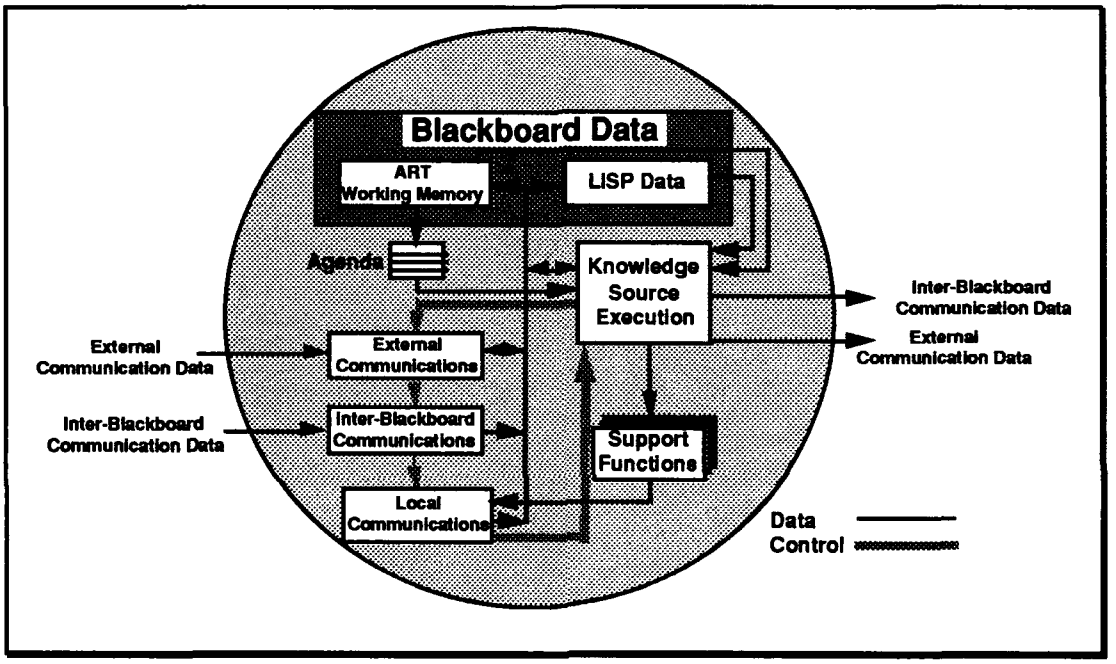

Figure 2.4 McAir's Module Internal Architecture

2.3.4. Adaptive Suspension Vehicle. The Adaptive Suspension Vehicle (ASV), is a self-contained six-legged, human operated, walking vehicle. The ASV consists of an engine driving 18 hydraulic leg actuators, over 100 leg position, leg velocity, leg pressure, and inertial sensors, and a laser range scanner [Payton, 1991:53] [Bihari, 1989:59-62]. The processing suite consists of seven Intel 80386 processors and two special purpose numerical processors. As of August, 1991, the control software was approximately 100,000 lines of Pascal source code.

The ASV planning and control hierarchy is shown in Figure 2.5. The human operator "flies" the vehicle with a joystick. The Planned Vehicle responds to the joystick commands by providing leg movements to fulfill them while simultaneously avoiding local obstacles. The Planned Vehicle sees the

Servoed Vehicle as a system of interacting legs. The Servoed Vehicle takes plans provided to the Planned Vehicle from the Motion Planner and sends the appropriate commands to each Servoed Leg. Each Servoed Leg knows its corresponding physical leg's state from the sensors mentioned above. The Servoed Leg translates the commands from the Servoed Vehicle into direct commands to the Physical Leg's three hydraulic actuators. [Payton, 1991:52]

The Servoed Legs and Servoed Vehicle operate in the a well-defined world and as such, functional as 'traditional real-time systems' as discussed in section 2.1. Extensive analysis of the physical legs being controlled resulted in a 10ms period for each Servoed Leg and a 25ms period for Vehicle Servo. Although the Leg Servo ideally runs at a 5ms period, *processing power limitations* forced a 10ms period and correspondingly slightly degraded performance. The Leg Servo can even miss an occasional period, allowing for some slight timing constraint negotiability. The rate-monotonic scheduling algorithm (section 2.1) was used to implement these fixed period, fixed priority tasks.

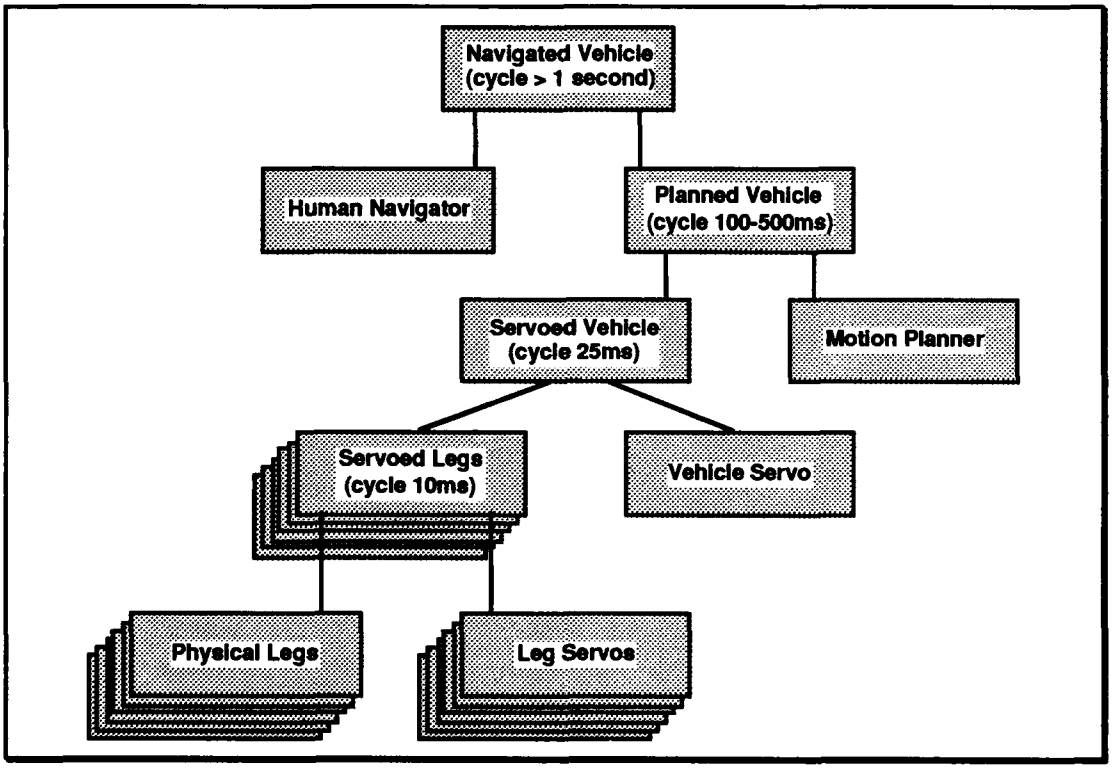

Figure 2.5 ASV Planning and Control Architecture [Payton, 1991:55]

The Motion Planner differs from the servo control in that it operates on the constantly changing terrain. It periodically produces a continuous velocity plan segment that is steadily consumed by the Servoed Vehicle. The plan segment is made up of two parts: a normal portion containing requested body acceleration to be used for a certain duration into the future (Pi_n), followed by a safety portion containing contingency deceleration to zero body velocity (Pi_s). Each newly generated plan segment overwrites the previous plan segment's safety portion. This two portion plan segment method operates as a 'forward recovery' technique in case of planning system failures as illustrated in Figure 2.6.

The Motion Planner's period varies between 100ms and 500ms depending upon the current state of the vehicle and the environment. Payton and Bihari state that the time to calculate a plan segment is approximately:

$$DTcp = \frac{Vmax/As}{SR - 1} + DTov \tag{2.5}$$

where

DTcp = the time required to compute a plan segment

Vmax = the vehicle's maximum allowable velocity

As = the acceleration during the safety portion of the plan segment;

SR = the rate at which the internal simulation of the vehicle's movements runs, as a multiple of real time;

DTov = a fixed amount of overhead time associated with computing a plan segment [Payton, 1991:56]

Although not currently implemented in the ASV, the existence of *Vmax* in the planning cost equation (2.5) could allow the scheduling algorithm to adjust its performance to meet timing constraints, telling the vehicle to slow down to allow for more planning time. The ASV does, however, use an adaptive scheduling algorithm. It starts with short planning periods and monitors how close to its deadline each segment completes. Payton and Bihari state that "If plan segments are completed uncomfortably close to their deadlines, it lengthens the normal portion of the plan segments and increases the planning period." [Payton, 1991:56].

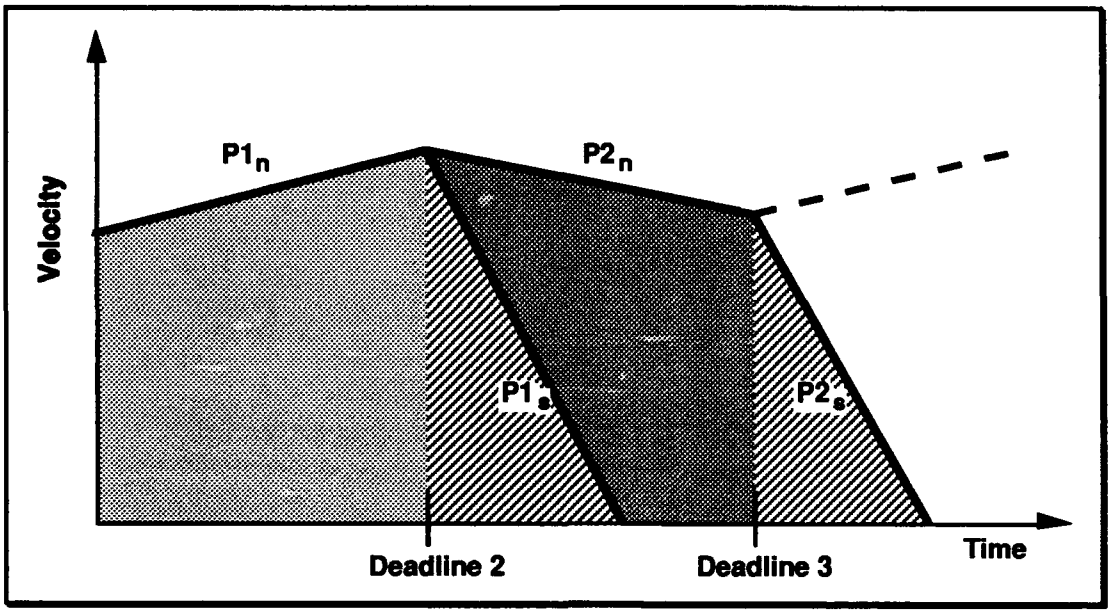

Figure 2.6 ASV Plan Generation [Payton, 1991:55]

In summary, because the six-legged vehicle can become unstable if the legs operate independently, the ASV fits nicely into the centralized, hierarchical control structure. The ASV has successfully integrated both a fixed rate-monotonic scheduling algorithm (controlling the leg servos) and an adaptive scheduler handling the planning tasks. The two-portion plan segment (normal and safety) technique is adaptable to a variety of systems. One can envision an airborne autonomous vehicle simply circling if the planner misses its deadline. Additionally, the link Payton and Bihari make between planning time and vehicle velocity is also adaptable to other domains. Both of these features should be addressed in any intelligent real-time system. However, this thesis is focusing on the software architecture issues of intelligent real-time systems, and not on domain-specific techniques.

2.4. Common Intelligent Real-Time System Components

In order to proceed with the development of an architecture to support intelligent real-time systems, it is necessary to determine what components appear to be common among the systems discussed so far. The architecture as developed in the Pilot's Associate program is perhaps a good place to start (Figure 2.2). In the figure, the top level components that have been identified are a Blackboard Process, an I/O Process, and a Reasoning Process. What is missing from this architecture is the notion of a real-time task scheduler or task manager. Basically, the McAir Pilot's Associate approach to real-time can be summed up by the term 'agenda-management'. Because there is no preemption, or guarantee of any kind on the execution time

of the KSARs, the approach needs to be enhanced to handle hard real-time constraints and provide some guarantees of performance.

The ASV goes one step further in that it does add a real-time task scheduler. It uses a real-time operating system, employing rate-monotonic scheduling to control execution of the leg actuators. The ASV also includes a component that performs the I/O Process similar to the Pilot's Associate, namely the component that sends commands to the leg actuators and receives data from the system's sensors. Additionally, it possess a reasoning process in the form of the motion planner. However, its architecture does not address the computation time – solution quality trade off issue.

In [O'Reilly, 1988], O'Reilly and Cromarty propose still another component of an intelligent real-time system called a metaplanner. The job of the metaplanner is to plan, monitor, and evaluate the execution of problem solving. O'Reilly and Cromarty state that such a system would also require a model of the computational times and cost of the various tasks the system can perform. With this 'system model' the metaplanner is able to make trade offs about which tasks to execute and when, given the current state the system finds itself in and the amount of time available to calculate a solution.

From all of these sources, I have drawn the conclusion that the following components need to exist in any intelligent real-time system architecture:

- *Environment Model.* This component performs the job of the blackboard in the Pilot's Associate program, acting as a repository of information about the current state of the environment external to the system.

- *System Model.* This component is used to evaluate potential threads of execution. It consists of a graph like structure which indicates the hierarchical ordering of the system's problem solving.

- *I/O Process.* This component translates signals from the environment into a format usable by the system and vice versa.

- *Reasoning Process.* The Reasoning Process or perhaps metaplanner, provides the adaptive control needed by the system to act 'intelligently'. Its primary job is to determine *what* the current task set is in contrast to solving the problems associated with the current task set.

- *Task Scheduler.* The Task Scheduler schedules the workload of the system based upon directions from the Reasoning Process, and provides some real-time guarantees of meeting deadlines.

2.5. Schedulable Task Types

I have also drawn the conclusion that in an intelligent real-time system there exist three broad categories of tasks: periodic, singular, and any-time. Each task also is broken into a mandatory part and an optional part. Mandatory task parts represent the minimum acceptable processing required in order to complete a task. Optional task parts represent the additional processing to increase a solutions quality. All mandatory task parts are fully deterministic while the optional task parts execution time depends upon the type of algorithm used.

The three task types allow for a mapping of the task types discussed in sections 2.1 and 2.2. Periodic tasks are real-time periodic tasks as described in section 2.1. Both the mandatory and optional parts of periodic tasks are fully deterministic. Singular tasks are non-periodic tasks that generalize the sieve function and primary/alternate task types in section 2.2. The optional part a singular task gives a more precise solution, but requires more processing time to compute. Any-time tasks generalize the monotone tasks in section 2.2. The optional part of an any-time task provides solutions whose quality increases with processing time.

2.6. Summary

Methods to determine the schedulability of periodic task sets are well defined. Methods also exist to schedule tasks which have non-deterministic execution times. Also, intelligent real-time systems are being developed that operate in a number of different domains. However, no intelligent real-time systems are currently addressing dynamic real-time task scheduling and control. The Pilot's Associate program has developed a system to dynamically adapt and control the execution of a knowledge-based system and is described in section 2.3.3. What the Pilot's Associate system is missing is a method of ensuring that tasks will meet their deadlines.

By integrating some of the traditional real-time systems methods, imprecise computation scheduling methods, and techniques developed in the PA and ASV programs, developing an intelligent real-time system architecture in Ada is feasible. The goal of this thesis research is to develop the architecture of such a system and demonstrate the feasibility of the design; in particular, demonstrate the ability to dynamically

create, schedule, and execute tasks to achieve real-time performance. Analysis of the results of this examination should allow one to suggest heuristics that can be used to dynamically determine the current scheduling policy.

III. Design Approaches, Assumptions, and Key Decisions

Chapter 2 provided some necessary background needed to design and implement an intelligent real-time system (IRTS). This chapter draws from that background information and provides design methodology for implementing such a system. In particular, how the architecture incorporates the common IRTS components identified in Chapter 2 is examined. In this chapter, performance measures for IRTS are examined along with design considerations that can be used to affect the performance measures. Following that, some basic design assumptions are stated and two possible design approaches discussed.

3.1. Performance Measures

Before attempting to design an architecture that supports intelligent real-time systems an examination of potential performance measures is prudent. Once these measures have been defined, it is then possible to begin making design decisions with an idea of the performance impact such decisions will have. The definition of performance measures then is a critical element in any design process.

A prevalent, but inaccurate, belief is that real-time systems are concerned with execution speed alone. In [Shamsudin, 1991], [Dodhiawala, 1989] and [Dodhiawala, 1988] the authors identify four performance measures for real-time systems. Those performance measures are as follows:

- *Speed.* This performance measure refers to the number of tasks executed per unit time. Speed is highly dependent upon the processing hardware. Generally, more and faster processors increase this performance measure. Additionally, efficient algorithms can increase the speed of a system.

- *Responsiveness.* "Responsiveness refers to the ability of the system to take on new tasks quickly. Operating in a rapidly changing dynamic environment, a responsive system perceives new developments early enough to compose and execute responses, possibly at the expense of ongoing tasks that may be delayed or even abandoned" [Dodhiawala, 1988:1–2].

- *Timeliness.* This measure "characterizes the system's ability to conform to task priorities" [Dodhiawala, 1988:2]. Assuming that not all tasks can be finished by their deadlines, a timely

system is one that finishes as many as possible. Thus some tasks must be "postponed, scaled down or discarded to allow the other work to be completed on time" [Dodhiawala, 1988:2].

- *Graceful Adaptation.* "This refers to the ability of the system to reset task priorities according to changes in the resource availability and/or demand and workload" [Shamsudin, 1991:15].

These performance measures are well suited for describing both traditional and intelligent real-time systems. However, two additional measures should be considered. The first is *data consistency* and the second is *solution quality.* Data consistency refers to the system's ability to maintain a timely and consistent view of the environment in which it operates. A timely view of the environment ensures that the system is solving problems that exist currently, and not responding to events that have been superseded by the passage of time.

A fundamental premise of intelligent real-time systems is that solution quality can be traded off against computation time to produce better solutions to the more important tasks. In order to determine the effectiveness of the system in performing this job, some measure of the system's solution quality must exist. This measure is probably the most domain specific of the performance measures mentioned thus far, but should be addressed by any evaluation effort.

Summarizing, the performance measures are speed, responsiveness, timeliness, graceful adaptation, data consistency, and solution quality. With the performance measures identified, the design effort can begin to focus on ways to affect those measures. In the next section, general design considerations are examined and related to the performance measures they are attempting to address.

3.2. Design Considerations

In [Dodhiawala, 1988:6] the authors presented a number of candidate methods useful when designing a real-time system. They suggest that a designer should consider *control reasoning, focus of attention, parallelism,* and improving *algorithm efficacy* when addressing design issues. Control reasoning is based upon the notion that knowledge about task demands, time constraints, system goals, and resource demands can be used to make smarter scheduling choices. Control reasoning can be used to affect graceful adaptation by recognizing overload situations and adjusting task priorities appropriately. Control reasoning can also impact the solution quality, timeliness, and responsiveness of the system. The limiting factor in applying control reasoning is the processing overhead associated with performing the reasoning and

implementing the controls. Temporal reasoning, speed/effectiveness trade-offs, supervisory control, and discretionary I/O are examples of implementable control reasoning strategies [Dodhiawala, 1988].

The capability to quickly respond to critical events is a desirable feature of any real-time system. This ability is referred to as the focus of attention. Preemption of executing tasks, prioritization of ready tasks, and the ability to change processing contexts quickly are methods of implementing focus of attention. Implementing focus of attention strategies directly affect a system's responsiveness and timeliness, generally with a corresponding decrease in speed as a result of processing the context switches.

Parallelism is primarily a concept associated with speed. By exploiting parallelism at both the architecture and application levels, significant performance gains are possible [Sawyer, 1990]. Multiple processors or concurrent processes are examples of methods to achieve parallelism. Increased parallelism generally results in increased design complexity. Adding parallelism to systems as an afterthought can be a significant challenge, since most of the inherent parallelism in the system may have been lost in the implementation.

Algorithm efficacy also primarily addresses the speed performance measure, but has a great impact upon the timeliness of the system as well. Methods used to improve algorithm efficacy include algorithm caching, incremental algorithms, and anticipatory processing. Generally, algorithm efficacy is an issue at design time, but each algorithm is tuned during implementation to meet system timeliness or speed requirements.

From the design perspective, each consideration needs to be addressed and each trade-off examined. First however, it is necessary to present the design assumptions that are a driving factor in this thesis effort.

3.3. Design Assumptions

The first assumption is that in any intelligent system, the majority of the work performed by the system is procedural in nature [Wilber, 1989:75]. For example, if one was to create a fully autonomous system to pilot a modern fighter aircraft, the existing computer software that drives the flight control, sensor, weapons, and navigation systems would in all likelihood not be replaced with AI programs. Rather, an AI system would be installed to control these existing systems, similar to the way in which a human pilot decides what system to activate, what mode to activate it in, and when to activate a particular system. Thus the AI system treats the other aircraft systems as either tools for gathering information or as effectors

to achieve a desired state. It does not duplicate the procedural functions already performed by the other systems.

The first assumption thus leads directly to the second assumption and the primary system design goal: design a system that allows an intelligent agent to control procedural tasks in such a way as to effectively and efficiently achieve a dynamically determined goal within real-time deadlines. The focus of the design effort then is not on the design of an IRTS for a specific domain, but rather on an architecture necessary to implement any intelligent real-time system.

An additional assumption is the use of Ada as the implementation language. Research and development communities, in my opinion, seem to believe that an implementation language is of little or no consequence to a research effort. In general this may be true, however, the rising trend in DoD software costs has resulted in a public law that dictates the use of the Ada programming language for all DoD software, making the implementation language an engineering issue. Any software system that is migrating into operational use must address the Ada implementation question. In essence, the implementing language is an engineering hurdle to overcome, similar to choice of the technology used to implement integrated circuits. This thesis effort recognizes the issue and attempts to add some basis for making decisions on the use of Ada in intelligent real-time system design.

The final design assumption stems from the belief that not all the desired processing can be done in the required time. The design assumption is that overload situations occur. If the opposite were true, then there is little need for any explicit control structure to prioritize what the system is doing. This design assumption dictates that the system must be able to prioritize the current task set to ensure those that are most relevant to the current situation are executed first *and* completed by their deadlines. Consolidated and restated, these four design assumptions are:

1) A real-time intelligent system consists of a large collection of procedural tasks whose execution timeline is determined by an intelligent agent.

2) The focus of this effort is on the software architecture of a system that provides the intelligent agent the necessary control to achieve its current goals in a timely manner.

3) The system must use the Ada programming language for implementation. Thus, it is critical to show how the concepts outlined in this thesis can be implemented in Ada.

4) The system is at times overloaded and thus must be able to dynamically prioritize the tasks. (Note that this thesis research does not purport to develop the domain knowledge necessary to determine what task is the most important at any particular instant; rather, it gives the intelligent agent in the system the tools with which to ensure those priorities are enforced).

Finally, a critical element of this thesis is real-time and, as discussed in Chapter 2, the *guaranteed* ability to meet deadlines. The design effort must provide a method for the system to ensure real-time performance as defined by the ability to guarantee deadlines. Additionally, it is important to remember that execution speed is always an issue, even if not explicitly mentioned, in any design decision.

3.4. Possible Design Approaches

Drawing from the background research, two fundamentally different approaches for achieving the goals of this thesis effort were envisioned. The first method is to modify an existing "reasoning process's" data and control structures to allow for the addition of real-time programming constructs. The second approach is to use standard Ada and design separate real-time data and control structures controllable by the reasoning process. The work done by the Pilot's Associate Program as outlined in section 2.3.3 seems to indicate that the first approach, modifying an existing reasoning process, would be the most appropriate. After a long period of consideration, I opted to use the second approach. What follows is a brief examination of the issues involved in both approaches and the reasons for the decision to not to attempt to modify an existing reasoning process.

3.4.1. Modifying CLIPS/Ada Design Approach. In section 2.3.3, the approach to achieving real-time performance in an intelligent system can be summed up by the term *agenda management*. The system must determine what is the most important knowledge source, process, rule, or other computation to execute next. Agenda management requires an agenda, and a method to prioritize items on the agenda. Thus for my purposes, I needed a reasoning process that allows for basic agenda management and, because of my third design assumption, is implemented in Ada. Additionally, since modifications to the source code of the reasoning process are practically guaranteed, access to the source code and documentation is required. CLIPS/Ada is the tool I found best suited to fill this role.

CLIPS/Ada is an Ada implementation of version 4.3 of NASA's C Language Integrated Production System. CLIPS/Ada provides a basic set of functions which can be used to achieve agenda management and rule prioritization [CLIPS-Ada, 1991:20] [CLIPSRefMan, 1991a] [CLIPSRefMan, 1991b] [CLIPSUG,

1991]. Given these basic building blocks, it is easy to envision adding the modifications as outlined in [Dodhiawala, 1988], [Lambert, 1990], and [Lambert, 1991] and discussed in section 2.3.3. In particular, adding channels or multiple agenda queues, scheduling policies based upon completion deadlines, and monitors to detect missed deadlines appears a straight-forward task.

CLIPS/Ada provides three basic parts to its rule structure as shown in Figure 3.1: the declaration part, the conditional-element(s) (the asterisk indicates plurality is possible), and the action(s). The conditional-element is traditionally referred to as the left-hand side of the rule and the action part is right-hand side of the rule. The declaration defines properties of the rule, the conditional-element(s) defines the conditions necessary for the rule to execute, and the action(s) defines what should occur when the rule executes.

```
(defrule <rule-name> [<comment>]
   [<declaration>]                    ; Rule Properties
   <conditional-element>*             ; Left-Hand Side (LHS)
=>
   <action>*)                         ; Right-Hand Side (RHS)
```

Figure 3.1 CLIPS Rule Definition Structure [CLIPSRefMan, 1991a:27]

The only currently defined declarative characteristic available in CLIPS/Ada is the *salience* of the rule. Salience is the priority of the rule and can either be assigned dynamically or statically. The salience defines which rule on the agenda of active rules executes next and thus provides one feature required of an intelligent real-time system. Additionally, because CLIPS/Ada allows the salience to be evaluated at every cycle through the rule execution loop, it is easy to imagine the agenda management functions discussed in section 2.3.3 being added with relative ease.

What is missing from the CLIPS/Ada system is the ability to determine and enforce the execution time allotted to each rule's actions or of the CLIPS/Ada inference engine itself. Thus, once a rule is chosen it is executed to completion, regardless of any changes in the environment. Additionally, the ability to declare periodic rules or functions (periodic tasks are fundamental in real-time systems) is also missing, although CLIPS version 5.1 is beginning to address this issue [CLIPSRefMan, 1991a:220]. Thus, for my

purposes, I would be required to add the ability to declare and execute periodic tasks, ensure execution deadlines, and interrupt or terminate executing rule actions.

One method of incorporating these abilities into CLIPS/Ada would be to expand the declarative part of the rule structure to include additional characteristics necessary to ensure real-time performance. Figure 3.2 shows an example of such a modification to the CLIPS/Ada rule structure. Here added declarations of rule-type, mandatory-duration, and optional-duration allow scheduling decisions to be made. Additionally, the RHS actions are divided into mandatory and optional actions which can be executed as determined by the current situation and scheduling policy.

```
(defrule <rule-name> [<comment>]
   [<declaration>                          ; Rule Properties
      <salience>
      <rule-type>
      <mandatory-duration>
      <optional-duration>]
   <conditional-element>*                  ; Left-Hand Side (LHS)
=>
   <mandatory-action>*                     ; Right-Hand Side (RHS)
   [<optional-action>*])
```

Figure 3.2 Modified CLIPS Rule Structure

Adding the ability to start, stop, and terminate executing rules is much more complex. First, it implies multiple processes operating concurrently. Each rule therefore needs to be encapsulated in a form that allows control of its execution time. Because my design assumes Ada is used and Ada concurrency is expressed in Ada tasks, essentially, each rule needs to become a separate Ada task. This is a significant change from the single thread of control CLIPS/Ada executes under now and a method to translate CLIPS/Ada rules into Ada tasks is required.

These changes are possible to implement; however, the job of implementing them is staggering [Sawyer, 1990]. First, the CLIPS/Ada source code contains over 130 files that total to more than 2.5MB of source code. Second, the changes involve both syntactic and semantic modifications, implying changes through out all the source code. Finally, this thesis effort is focusing on the design of an generalized architecture capable of supporting IRTSs, and not restricted to only rule-based systems; thus, this approach

was deemed currently impractical. The modification of CLIPS/Ada to support real-time is left as future research.

3.4.2. Controllable Real-Time Task Manager Approach. The second approach consists of using standard Ada and designing a system that supports real-time constraints and is controllable by another process called the Reasoning Process. The goal is not to modify the Reasoning Process itself, but rather to treat it as a separate entity that controls a real-time system. The Reasoning Process must be able to control what the current task set is, what the current scheduling policy is, and be able to change them as required. What it is not required to do is the actual scheduling and execution control of each task in the currently defined task set. This elevates the Reasoning Process to a "supervisory level", acting as the central element in the control reasoning of the system, and alleviated of the bookkeeping necessary to implement the desired scheduling policy.

This approach requires a separate entity called the Task Manager that controls the instantiation, scheduling, and execution of Ada tasks as directed by the Reasoning Process. Additionally, it provides status information about the state of a task or the state of the system as a whole to the Reasoning Process. The design process using this approach requires that a set of controllable parameters be devised that are manipulated by the Reasoning Process and implemented by the Task Manager. *The identification and implementation of these controllable parameters is the heart of this thesis effort.*

As implied above, the controllable parameters can be divided into groups based upon the function they are performing: scheduling policies, control variables, status variables, and task controls. Example scheduling policies are earliest deadline first, important task first, and shortest task first. Example control variables are the percentage of CPU time to allocate for periodic tasks, or execute mandatory or optional task parts flags. Example status variables include the actual number of tasks currently running or the percentage of CPU time currently in use. Finally, task controls include task instantiation, task modification, and task deletion. Note that creating a task is a different operation than scheduling a created task.

The concepts and methods developed using this second approach are also applicable to the first approach. It is conceivable that the Task Manager can be incorporated directly into the Reasoning Process. Alternatively, one can choose from the methods developed here to meet real-time constraints and only incorporate those that seem to provide the most utility. Taking this second approach does not prevent someone from taking the first approach and may actually be the necessary first step in the first approach.

The remainder of the chapter addresses the design decisions made to use the second approach and concisely defines the design problem.

3.5. Design Problem Statement

A statement of the design problem now helps to define what this research is attempting to do. It is important to remember that an underlying assumption of this thesis effort is the existence of a domain specific problem solving structure (plan-goal graph or task network). Remembering that, and drawing upon the information presented thus far in this thesis, the design problem can be stated as follows:

> Given a graph structure (plan-goal graph or task network) that represents the system's operation, and some currently active subset of that structure, dynamically generate and execute a processor schedule to maximize system performance based upon a dynamically determined scheduling policy. Each node in the graph represents a task to accomplish in response to some event from the environment or needed operation to accomplish the system's mission, and each node represents either a periodically performed task, single event response task, or a continually refined event response task (an any-time task).

3.6. Key Design Decisions

From the background research performed, constraints upon development tools, and my own intuition, a number of key design decisions are easily made. This section explicitly lists those decisions and some of the reasons for each one. In short, those key design decisions call for small scale parallelism, asynchronous operation, Ada tasks as the scheduling elements, dynamic Ada task instantiations, and a single processor feasibility implementation.

All of the systems examined in the background research were implemented with between four and eleven processors [Lambert, 1990] [Aldern, 1990] [Payton, 1991]. This relatively small number of processors can generally be classified as small scale parallelism. The architecture developed in this thesis effort should consider a partitioning of functions suitable for a small number of processors.

Asynchronous operation of the partitioned functions is also a conclusion drawn from the background research. Each of the partitioned functions should operate at its own speed as dictated by the work it performs. This approach allows for decoupling of the system components and corresponding performance

benefits. Additionally, events external to the system, generally asynchronous in origin, can be handled as they occur.

A direct result of the design assumption of Ada as the implementing language is the use of Ada tasks. Ada represents concurrency in the form of tasks and thus any system implemented in Ada, and intended to execute concurrently, should use Ada tasks. This implementation language imposed constraint must be addressed if task scheduling is to be performed in this system.

A decision resulting from my literature review is the requirement for dynamic Ada task creation and control. From examination of knowledge engineering documents of the Pilot's Associate program, it is clear that in any real system, there are a very large number of tasks the system can perform. Clearly, not all of the tasks can be economically instantiated simultaneously within the limits imposed by today's processing systems. This forces me to the conclude that tasks must be created or instantiated dynamically at run time.

Finally, the issue of a design feasibility demonstration must be addressed. Given the tools available for this research effort, and the limited time to develop a demonstration, a single processor feasibility demonstration is the only viable alternative. Since, this thesis effort is the beginning in a series of related efforts to develop and flush out an intelligent real-time system, I believe the most important issue to address first is the feasibility of dynamic task creation and scheduling. For that reason, I have chosen to start first with a single processor implementation, while maintaining concurrency in the design.

3.7. Design Approach Summary

This chapter has attempted to provide some of the design methodology and some design guidelines used in developing this intelligent real-time system architecture. In summary, I have listed some performance measures and some design concepts and methods to affect those performance measures. Additionally, basic design assumptions and some key design decisions have been addressed. The following chapters in this thesis develop an architecture based upon the design assumptions and decisions, and examine the issues arising from that development effort.

IV. An Intelligent Real-Time System Architecture

Because the scope of this thesis investigation is so large, the research and development effort is not completed within this thesis effort. The research area of intelligent real-time systems is diverse and complicated, and this effort is beginning the process of examining the issues involved. Examination of the issues is done by incrementally designing and implementing a system capable of performing the role of an intelligent real-time system (IRTS) and addressing potential problems as they arise. Acknowledging the broad subject area of this research, it is necessary to provide a larger vision of how such a system should be constructed. The goal of this chapter is to provide that vision.

4.1. Top Level Design

The top-level intelligent real-time system conceptual architecture is shown in Figure 4.1. Briefly, the Environment Model acts as the data repository, similar to a blackboard as discussed in section 2.3.3. The I/O Process is responsible for communications with the environment. The Reasoning Process is responsible for determining the currently active set of tasks and the current scheduling policy to use in scheduling that set of tasks. The Task Manager is responsible for implementing the scheduling policy and notifying the Reasoning Process of the current status of the task set. Finally, the System Model represents the system's problem solving approaches in a form that can be used by the Reasoning Process in determining the currently active task set. The top-level design includes all the basic components of an intelligent real-time system as identified in Chapter 2. The Reasoning Process is filling the intelligent agent role and uses the Environment Model, System Model, and status information provided by the Task Manager to reason with. The Task Manager schedules tasks as directed by the Reasoning Process to achieve real-time performance.

The architecture vision presented in Figure 4.1 attempts to maintain as much parallelism as possible in the design. From the background research, it can be concluded that most IRTSs run on multiple processors, usually loosely coupled or distributed. The recognition of that probable direction is shown by separation of system functions. Each function is envisioned as a separate Ada task. As mentioned previously, in Ada, parallelism is represented in tasks. Additionally, the language provides a number of features specifically designed to handle task scheduling and inter-task communications [Booch, 1983:231–

304] [Locke, 1992] [AdaLRM, 1983:Chapter 9] [Real-Time, 1984]. Methods other than using Ada tasks are possible and should be examined, but for the purposes of this effort and the top-level design, Ada tasking is a keystone.

The component most lacking in previous intelligent real-time systems is the Task Manager. For that reason, it is the primary focus of the rest of this thesis effort. However, before discussing the design issues of the Task Manager, the other components are briefly discussed. The discussion also contains suggested approaches to implementing the components not yet implemented in this research effort.

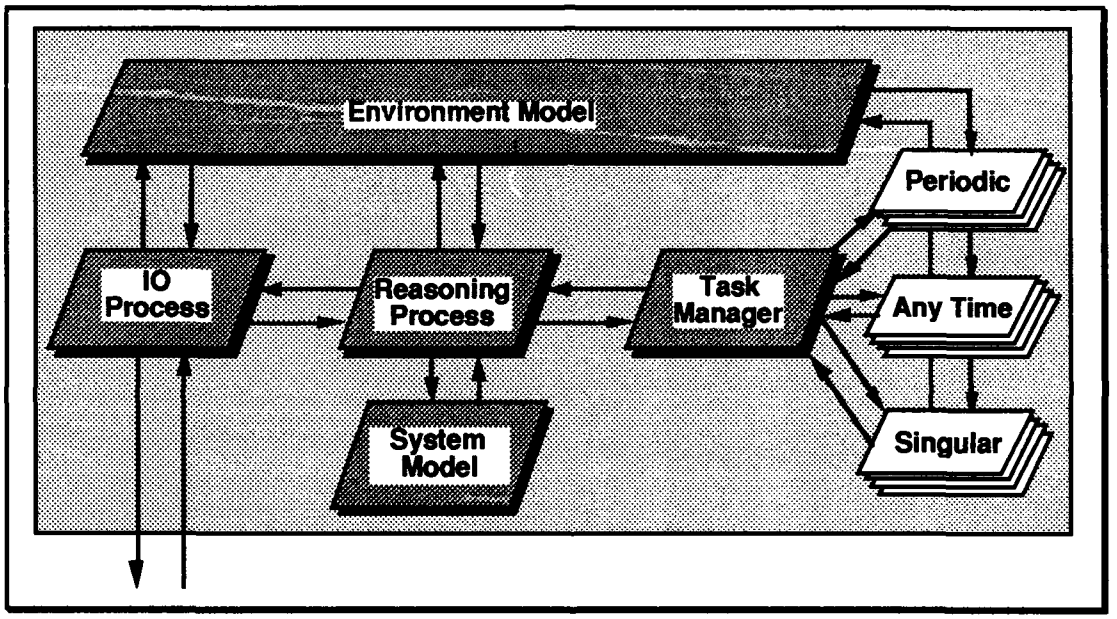

Figure 4.1 Top Level Design Diagram

4.2. Environment Model

The development and implementation of the Environment Model is beyond the scope of this particular thesis effort. However, its impact on the overall system design must be accounted for and addressed in any feasibility demonstration. The Environment Model is perhaps the most domain specific aspect of any IRTS. The Environment Model encapsulates all the data structures that are deemed relevant to the system's ability to function intelligently in the environment. The exact method used to represent the environment model is an open issue; however, this research plans to pursue an object oriented approach.

Along those lines, my investigations have centered around the use of Classic-Ada to implement an object-oriented environment model [ClassicAda, 1989]. This package provides all the standard object oriented data and control structures. In addition, it allows object references to be passed by means of an object ID value. This ability is highly desirable when passing information to the Task Manager for use in controlling tasks. Essentially, a pointer task's data structure can be passed and stored as a single object ID. Although shared memory structures are common practice in real-time systems, there is a potential communication bottleneck that can be created with this approach. Future efforts should examine this issue closely.

Again, it is out of the scope of this thesis effort to fully implement an environmental model. Identifying the relevant ideas (passing pointers to data structures by object ID and object oriented design techniques) suffices to give the direction this is pursuing. Additionally, it provides general guidance on where to begin in any follow-on thesis effort.

4.3. System Model

Development of a specific system model for a particular domain is not the purpose of this thesis. As has been mentioned previously, this research is predicated upon the assumption that a System Model for the chosen domain already exists. Thus for the purposes of this architecture, a method of representing that model needs to be provided. The planned direction is to use a graph theory approach to represent the System Model.

It is important to point out that the graph structure used to represent the system to the reasoning process is not, in fact, the system. Instead, each node in the graph is a representation of some processing required of the system as a whole. The actual code that represents each node in the System Model is accessed and controlled by the Task Manager.

While the system is executing, there should conceptually be two system models present. The first one is the sum total of all the possible nodes that the system can instantiate, and the second represents the set of tasks currently instantiated by the system. The first model contains in each node, information on how to instantiate that particular node and the real-time impacts of instantiating that node. The nodes in the second network should contain the information actually used to instantiate that particular instance, and information that allows the Reasoning Process to communicate with the instantiated task.

Simple graph functions should be able to accomplish the job of the System Model effectively. Thus, the design must allow one to add or delete nodes in the graph, search the graph for a particular node, and allow the reasoning process access to the data stored in that node. All of these are standard graph functions available as Booch components [Booch, 1986]. In addition, reasoning about the temporal relationships between tasks must be allowed for. Future research should pursue the use of temporal logic and temporal constraint networks to achieve that ability [Wood, 1989] [Dechter, 1991].

4.4. I/O Process

The exact internal workings of the I/O Process is again implementation dependent. The important issues to address are the fact that I/O is required and the relative priority associated with I/O in general. For example, reducing sampling rates on I/O channels can be one method for the Reasoning Process to gain some CPU time. However, critical external signals must still be acknowledged and dealt with.

The concepts of periodic and non-periodic tasks with varying importances and execution priorities can be easily extended to include I/O processes. This implies that the Reasoning Process could treat the I/O Process as simply another task to be scheduled and executed by the Task Manager. The purpose of making an explicit I/O Process in the overall design is to emphasize the fact that communications with the world external to the system is inherent in all intelligent real-time systems.

4.5. Reasoning Process

Although the focus is on the architecture required for an intelligent real-time system, it is primarily addressing the controls necessary and not the design of the intelligent agent that manipulates the controls. It should not be taken as a given that the appropriate choice for an intelligent agent is always a rule based expert system. Rather, the nature of the intelligent agent should be viewed as domain dependent. Possible choices for an intelligent agent include straight procedural code designed for the particular domain, a rule based expert system, or a neural network or connectionist approach. It is my belief that eventually, the intelligent agent should incorporate all of these approaches.

However, this thesis effort is primarily addressing the Task Manager and some compromises have to be made. For this reason, the Reasoning Process used is a rule-based system made up of CLIPS/Ada and the appropriate interface code. CLIPS/Ada was available and provided the necessary functions with which to prototype the system operation. In addition, both the source code and user manuals were available.

The basic approach is for the I/O Process to signal the occurrence of events in the environment by asserting facts into the CLIPS fact base. CLIPS rules are then pattern matched against these facts and actions taken as appropriate to guide the system into a desired state. Those actions can include directing the Task Manager to create new tasks, modify current tasks, or remove current tasks. The Task Manager responds to these actions and asserts new facts into the CLIPS fact base that describe the state of the currently active task set. CLIPS rules are again pattern matched against these new facts and further actions taken as appropriate to handle any problems. CLIPS is filling the role of a metaplanner for purposes of this design.

4.6. Task Manager

The literature review and analysis of intelligent real-time systems conducted in Chapter 2 revealed three broad categories of tasks the Task Manager needs to manage. These categories are *periodic*, *singular*, and *any-time*. A periodic task arises from the necessity to either control some process or monitor for some condition. Periodic tasks are fundamental to traditional real-time systems as exemplified by the large body of knowledge covering real-time periodic task scheduling [Sprunt, 1989] [Sha, 1989] [Broger, 1989] [Borger, 1989] [Sprunt, 1990] [Sha, 1991] [Lamont, 1991]. Examples that produce periodic tasks are things like maintaining a flight path in the presence of strong winds, or monitoring a radar track to ensure it does not become a threat, or screen updates to video displays.

Singular tasks and any-time tasks are both types of non-periodic tasks. Examples that produce singular or any-time tasks are things like taking off in an autonomous aircraft, or responding to an obstacle that blocks the path of a mobile robot. In the autonomous aircraft, taking off usually only happens once during the life of the mission and once accomplished, that task can be forgotten and purged from the active task list. The mobile robot may encounter many obstacles while negotiating a route or may not encounter any at all. In either case, there are generally specific task starting times, execution sequences, and deadlines that must be adhered to.

The difference between singular and any-time tasks lies in the underlying algorithm encapsulated by the task. A singular task arises from the asynchronous nature of events occurring in the real-world, or one time steps in a control process. An any-time task is similar to singular task, in that it arises from the same conditions, however, its solution method differs significantly. A singular task has a specific starting time

and after some time interval provides an answer. An any-time task also has a specific starting time, but provides answers of increasing accuracy (or quality) the longer the task is allowed to run.

Additionally, one of the premises stated for an intelligent real-time system is the ability to trade-off solution quality and execution time. Thus, each task is assumed to have a mandatory and optional part. This concept, as outlined in section 2.2, provides one method of performing the solution quality/computation time trade-off and is adopted fully in this approach. The mandatory part of each task ensures that, at least, some answer is provided by the intelligent real-time system by the task's deadline. The optional part of each task provides ways to increase solution quality, provided execution time is allotted.

Finally, a method of mixing the task types together in one system needs some consideration. Real-time systems generally cast both periodic and non-periodic tasks into the periodic framework [Sprunt, 1989] [Sprunt, 1990]. Using this approach, non-periodic tasks are given periodic time slices in which to execute that go unfilled if not needed. The approach taken in this research is a slightly different methodology that allows the Reasoning Process to control the amount of CPU time allotted to periodic tasks, called the "budgeted periodic utilization". Non-periodic tasks are scheduled to execute in the remaining CPU time.

The impact on task scheduling of these varying task types is enormous and is discussed in greater detail in the following sections. However, the need to achieve real-time performance levies a common set of constraints upon how each task type is dealt with. In particular, it is necessary to determine if a given task set is feasible (i.e., can be scheduled to meet its deadlines) and then to schedule the task set. Also, since it is assumed that there are infeasible task sets, an infeasible task set needs to be detected, and a feasible sub-set of the tasks scheduled.

4.6.1. Periodic Task Scheduling. Periodic tasks have a period, a mandatory and optional duration, and a dynamically assigned *importance* which represents the current relevance of the task. The distinction between importance, and priority is necessary to emphasize. The *priority* of a periodic task is the value assigned to the task by the scheduler to ensure the task's execution characteristics. The task's importance is a system wide value of the task's relevance to the current problem. The problem for the periodic task scheduler then is to assign an execution priority to a task, given the task's importance, a scheduling policy, and the current state of the system.

The rate-monotonic theory plays a critical role in fulfilling the job required of the periodic task scheduler. As discussed in section 2.1, the rate-monotonic algorithm provides a method of determining the feasibility of a given task set and a corresponding priority assignment method for a feasible task set. By keeping track of the durations and periods of the periodic tasks, it is simple to see if an additional task can be accommodated with a guarantee of its ability to meet its deadline. Table 4.1 below is an example set of periodic task that is used to illustrate some of the potential methods to schedule periodic tasks.

Table 4.1. Example Periodic Task Set

Task	Desired Period	Max Period	Mandatory Duration	Optional Duration	Total Duration
T_1	33	50	1	1	2
T_2	67	100	1	3	4
T_3	67	100	2	4	6
T_4	100	100	2	5	7
T_5	360	450	4	6	10
T_6	500	750	4	8	12
T_7	750	1000	3	15	18
T_8	1000	1500	3	11	14

From Table 4.1, one can determine four possible periodic utilizations; (1) minimum mandatory utilization, which is the absolute minimum periodic utilization that can be achieved, (2) maximum mandatory utilization, (3) minimum optional utilization, and finally, (4) the maximum optional utilization, which is the absolute maximum periodic utilization of the current task set. Figure 4.2 gives the relative positions of the four values.

Note that there is no minimum period given for each task in Table 4.1. The desired period is assumed to be the minimum period. It is inconsistent with real-time system design to propose periodic task

sets whose periods are not the minimum desired. This implies then that the periodic scheduler does not try to maximize processor utilization simply because processing time may be available. Rather, the periodic task scheduler tries to obtain the best solution quality (as a function of optional task portions scheduled and executed) based upon available time.

Assuming the current set of periodic task consists of tasks $T1$ thru $T7$, the maximum utilization of the example periodic task set is shown below. Note that the maximum utilization calculation uses the minimum, or desired periods and both the mandatory and optional task durations.

$$\begin{aligned} Util_{\max} &= \sum_{i=1}^{n} \frac{(m_i + o_i)}{p_i} \\ &= \frac{(m_1 + o_1)}{p_1} + \frac{(m_2 + o_2)}{p_2} + \ldots + \frac{(m_7 + o_7)}{p_7} \\ &= \frac{2}{33} + \frac{4}{67} + \frac{6}{67} + \frac{7}{100} + \frac{10}{360} + \frac{12}{500} + \frac{18}{750} \\ &= 0.3673 \end{aligned}$$

where m_i and o_i are the mandatory and optional durations of each task and p_i is the period of each task in the set. And similarly, the minimum utilization is calculated using the maximum periods and only the mandatory part of the tasks.

$$\begin{aligned} Util_{\min} &= \sum_{i=1}^{n} \frac{m_i}{p_i} \\ &= \frac{m_1}{p_1} + \frac{m_2}{p_2} + \ldots + \frac{m_7}{p_7} \\ &= \frac{1}{50} + \frac{1}{100} + \frac{2}{100} + \frac{2}{100} + \frac{4}{450} + \frac{4}{750} + \frac{3}{1000} \\ &= 0.0872 \end{aligned}$$

Equation 2.1 says that with 7 tasks, the utilization must be less than or equal to 0.728. However, in our case we are restricting the CPU time allotted to periodic tasks to not exceed the current budgeted periodic utilization. Thus Equation 2.1 must be adjusted to reflect this further restriction as follows:

$$\sum_{i=1}^{n} \frac{(m_i + o_i)}{p_i} \leq n\left(2^{\frac{1}{n}} - 1\right) BU \qquad (4.1)$$

where BU is the budgeted periodic utilization. Assuming the current budgeted periodic utilization is set at 0.51, we get a required utilization of 0.728 * 0.51 = 0.3712. Since the current maximum utilization is less than 0.3712 we can guarantee that the deadlines of our periodic task set are meet.

Adding *Tg* to the task set raises the utilization to 0.3696. Using (4.1) the required utilization with 8 tasks and a budgeted periodic utilization of 0.51 is 0.3693, and we can no longer guarantee the ability to meet the deadlines for the task set. However, we can still guarantee the deadlines of the task set if only the mandatory parts of each task are executed, or the budgeted periodic utilization is raised. It is the job of the Task Manager to schedule the task set as best it can and notify the Reasoning Process of its current status. It is up to the Reasoning Process to either step in and adjust the task set, or let the Task Manager schedule the task set.

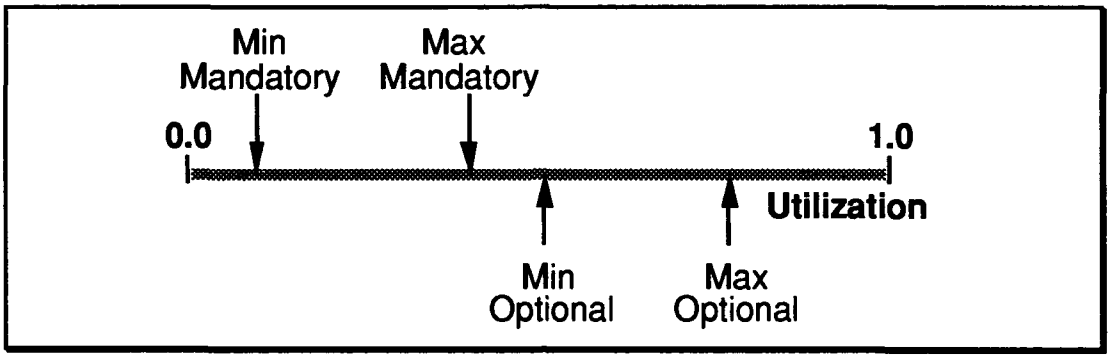

Figure 4.2 Some Calculable Periodic Utilizations

Using the minimum utilization, the maximum utilization and a dynamically determined budgeted utilization, the scheduler should initially schedule tasks using the rate monotonic algorithm with both the mandatory and optional portions of each task. When the currently budgeted periodic utilization is exceeded, then a scheduling policy decision must be made, and that decision is left to the Reasoning Process. From the periodic scheduler's point of view, it simply needs to detect a violation of the current policy, and be able to implement a new scheduling policy.

Some intuitively obvious scheduling policies are (as mentioned above) scheduling both the mandatory and optional task portions, or scheduling only the mandatory portions of tasks. Another task scheduling policy that directly affects the system's ability to degrade gracefully is *importance-ordered scheduling*. In importance-ordered scheduling, periodic tasks with higher importance need to be scheduled at a higher execution priority then tasks of lower importance. Note that this approach is directly addressing the issues of focus-of-attention and control reasoning.

The policy of scheduling both the mandatory and optional task portions can be viewed as the 'normal' case. Here the scheduler is trying to provide the best possible answer to every problem. When computation time is in short supply, switching to mandatory-only task portions frees up the maximum amount of processor time, at the cost of solution quality for all periodic tasks. When computation time is of a minimum but solution quality is important, switching to an importance ordered schedule is desirable.

In summary, the periodic scheduler needs to assign execution priorities to a new task, based upon the current scheduling policy. Thus, a scheduling step occurs whenever a new task is created. Additionally, the periodic scheduler needs to provide the reasoning process with a variety of implementable scheduling policies that achieve different goals. As mentioned above, these implementable scheduling policies should include at least, mandatory only, optional and mandatory, and importance-ordered.

4.6.2. Non-Periodic Task Scheduling. Non-periodic tasks consist of a mandatory and optional duration, a dynamically determined importance, a start time, a deadline, and possibly precedence constraints. Additionally, non-periodic tasks can either be singular or any-time tasks. The job of the non-periodic task scheduler then is to assign execution priorities to both singular and any-time tasks to ensure tasks are completed on or before the task's deadline.

The concepts used here are primarily derived from the work done by Liu, et al, [Liu, 1991]. Thus, as is the case with all tasks in this system, each task is assumed to have a mandatory portion and an optional portion. Again, as is the case with the periodic tasks, the job of scheduling non-periodic tasks can be decomposed into determining the feasibility of scheduling a task set, and actually scheduling the tasks.

Before discussing the proposed scheduling methods, a word about the durations of the optional parts of singular and any-time tasks is required. The optional duration of a singular task is a one time chunk of time that the task needs to compute a solution that is more accurate than its mandatory part. The optional duration of an any-time task is the amount of time it takes to cycle once through the solution refinement process. Thus when a singular task's optional duration has elapsed, the task is completed. When an any-time task's optional duration has elapsed, it is simply placed back into the set of tasks ready to execute until its deadline has passed or some other terminating condition met.

One is cautioned that the problem of optimally scheduling varying length tasks with precedence constraints preemptively is an NP-complete problem [Coffman, 1976:Table 1.1]. A real-time system that faces such a problem must make some simplifying assumptions, or place artificial constraints upon the

problem size. The approach taken in this thesis is to make some simplifying assumptions. Thus, optimal solutions are not generated; rather a working scheduling is found quickly. This approach means that the Task Manager does the on-line scheduling of the non-periodic task set *using* the assigned importances, deadlines, and start times while the Reasoning Process does the off-line scheduling to determine what the task set, importances, deadlines, and start times should be.

The scheduling policy used is, basically earliest deadline first as enhanced for on-line scheduling by Baruah, et al, [Baruah, 1991]. In addition, the earliest deadline first algorithm has been enhanced to deal with the concept of mandatory and optional task parts. The algorithm assumes the existence of two priority queues, one with tasks arranged with the earliest deadline first, and another arranged with the latest start time first. The latest start time of a task is its deadline minus its mandatory duration if its mandatory part has not completed, or its optional duration if it has.

When a new task arrives in the system, it is placed into the deadline and latest start time queues. The scheduler removes the first task from the deadline queue and begins to execute it. If this is the first time that task has been started, then it will be executing its mandatory part. When the task is completed, it is removed from the latest start time queue, its latest start time for its optional part is calculated and the task is reinserted into both the deadline and latest start time queue. As long as the system is not overloaded, the latest start time of any task will never be exceeded. An overload situation is detected by the presence of a task on the front of the latest start time queue whose latest start time is equal to the current time.

When an overload situation occurs, tasks are executed based both on the deadline of the task and its importance. The choice of which task to execute must now be made. Assuming that the goal of the system is to execute as many mandatory tasks as possible, then if the currently executing task is not executing its mandatory part, and the task with a latest start time equal to the current time has not executed its mandatory part, the currently executing task is preempted, and the other task is started. If both tasks are executing their mandatory parts, then the currently executing task is checked to see if it has any slack time. If it does, then again it is preempted and the other task is started. If it does not, then the task with the highest importance is allowed to execute and the other task misses its deadline.

Using the latest start time queue allows the scheduler to easily detect overload situations. Organizing tasks with both a mandatory part and an optional part allows the solution quality/computation

time trade off to be made. Combining the two allows one to define multiple scheduling policies that can be adapted to the current situation.

The "feasibility test" consists primarily of determining if every task in the task set can at least complete its mandatory part by its deadline. Given the task's mandatory duration, feasibility testing is fairly straight-forward with one exception. That one exception deals with accounting for the periodic task set's processor utilization. Figure 4.3 below illustrates the approach used to address this issue.

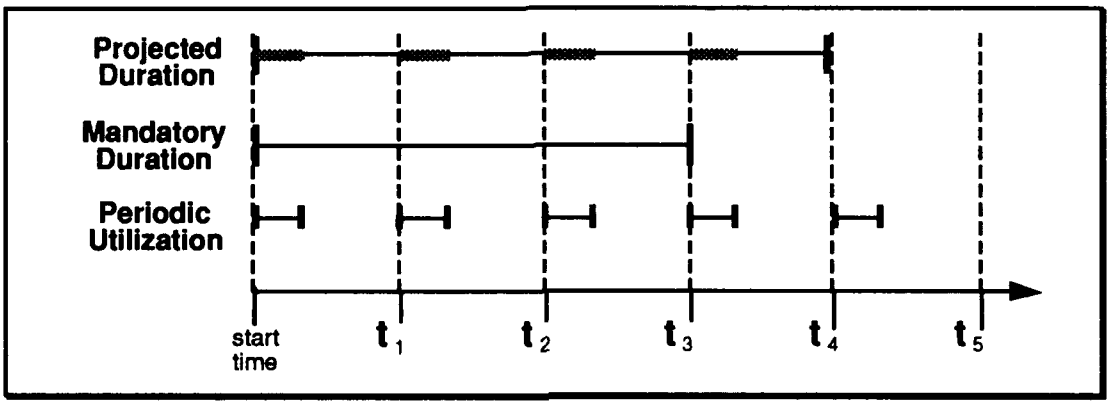

Figure 4.3 Predicting Non-Periodic Task Actual Durations

Since the periodic utilization is the percentage of the CPU time that is unavailable for non-periodic tasks, each non-periodic task's predicted execution time must include some amount of time waiting for the periodic tasks to complete. The approach taken is to simply divide the task's mandatory or optional durations by the processor utilization not used by the periodic tasks to predict the non-periodic task's actual duration. Thus, the feasibility of a particular non-periodic task is determined by the equation;

$$\forall_{i \in T},\ t_n + \frac{m_i}{(1 - pu)} \leq d_i \qquad (4.2)$$

where:

T = set of non-periodic tasks

d_i = deadline of task i

m_i = mandatory duration of task i

pu = current periodic utilization

t_n = time now

This method does have some problems, namely that the periodic utilization is an average value and not a accurate reflection of any one particular duration. System testing has to examine this issue to determine the validity of this approach.

In summary, both the periodic and non-periodic schedulers need to determine the feasibility of a given task set and assign execution priorities as directed by the current scheduling policy. Scheduling policies implemented by the schedulers should include both policies for feasible task sets and importance ordered scheduling for infeasible task sets.

4.7. Architecture Summary

The architecture as described in this chapter is a vision of how an intelligent real-time system should be constructed. Research into some of the components has already been accomplished (as is the case of the Reasoning Process by the Pilot's Associate program and outlined in Chapter 2). To add some validity to the approach outlined in this chapter, a feasibility demonstration of the architecture is necessary, and in particular, the functions of the Task Manager. This demonstration is described in the next chapter.

V. Feasibility Demonstration System

This chapter discusses in detail how the feasibility demonstration system is implemented. The primary focus of the feasibility demonstration is on the development of the Task Manager. As mentioned previously, the Task Manager is responsible for guaranteeing real-time performance of the system in normal load conditions. In addition, it must respond appropriately to changing task importances and task loads, and guarantee deadlines of a sub-set of tasks in overload situations. This chapter begins by discussing some general implementation issues, then gives a brief description of how each of the other architecture components is implemented. Following that, the data structures used in the Task Manager are examined along with the transitions each task type can make. Finally, the procedures that make up the Task Manager are reviewed.

5.1. General Implementation Issues

As with any demonstration effort, there are some implementation assumptions. In this effort, those assumptions are as follows. First, the number of processes that are running at any one time is unknown. This implies that the size of the problem is dynamic and should not be artificially constrained by some maximum size limit. Dynamism at the implementation level usually means variable-sized data structures. In this thesis effort, this translates to linked list structures, at cost of processing time and determinism. Arbitrarily deciding up front, the size of the problem would greatly increase the speed of the system, essentially reducing a large number of O(n) linked list operations to O(1) array operations. Future work should look at using predetermined data structure sizes and the performance issues associated with them.

Additionally, this thesis effort is unfunded and does not have primary access privileges to the computer system the design is implemented upon. What that means is there are always tasks running in the background, unknown to the Task Manager, that consume processor time. Thus, any testing of the Task Manager's timing characteristics are subject to an unknown amount of error. The research has not specifically addressed that issue in the scheduling algorithms used in this feasibility demonstration. However, the impact of the additional, unknown processor loading is missed deadlines, and missed deadlines are important events the system is designed to detect and handle.

5.1.1. Ada Compiler Choice. The choice of an Ada compiler plays a crucial role in this feasibility demonstration. Since Ada tasking is a "big" part of the thesis effort, the compiler's support of tasking primitives is an important issue. Chapters 3 and 4 expressed the need for dynamic priority assignments and any Ada development environment used needed to support them. The feasibility demonstration uses the Verdix Ada development environment over Meridian Ada for the following reasons.

First, Verdix Ada provides tasking primitives that allow direct control of the a task's priorities and execution. Verdix provides the procedures Set_Priority, Suspend_Task, and Resume_Task [VERDIX, 1990]. In addition, Verdix's has implemented the Priority Inheritance Protocol in their Ada run-time system. The Priority Inheritance Protocol prevents a major problem with Ada tasking known as priority inversion [Broger, 1989]. It is important that these parts are identified here because they are compiler specific and not readily transportable to other Ada development environments.

These specific procedures, Set_Priority, Suspend_Task, and Resume_Task all take the Verdix Ada defined Task_ID as input. Each procedure also returns a value that specifies the result of that particular operation. In the cases of Suspend_Task and Resume_Task, the returned values are used to determine whether or not a periodic task has missed its deadline, or if a problem exists with a non-periodic task. Also, the range of task priorities that are available with the Meridian compiler is 20, while the Verdix compiler allows for 100 different priorities and thus a finer priority resolution.

Additionally, Verdix Ada is available on most of the computer systems at AFIT and should continue to be in the future. Finally, both CLIPS/Ada and Classic Ada (an object-oriented Ada pre-processor I planned to use in developing the Environment Model) successfully compile with the Verdix Ada but not with the Meridian Ada compiler [ClassicAda, 1989]. The choice of Verdix Ada thus allows the inclusion of existing tools, provides readily transportable code within AFIT, eases the task control problem, and allows a greater range of task priorities. For these reasons, Verdix Ada was chosen as the Ada development environment.

5.1.2. Memory Management Issues and Impacts. Another implementation issue to be addressed in any real-time system is memory management. This means two things: first, ensuring that the memory does not become fragmented and thereby forcing a garbage collection problem; and second, ensuring that available memory is used efficiently. For these reasons, all the data structures used in this feasibility demonstration employ their own memory management. This means that once an item of that type has been

allocated from system memory, it is retained and reused by the data structure as needed and never returned to the system's heap space.

The issue of memory management has a profound affect upon how tasks are used, created, and deleted, presenting a significant challenge in the development of this feasibility demonstration. The actual work of the system is assumed to be performed by the periodic, singular, and any-time tasks (see Chapter 3). In Ada, any task object that is declared within the scope of another task only gives up its memory when the task that declares it terminates [AdaLRM, 1983:9] [Cohen, 1986:699-708]. Additionally, the language specifically prevents a user-directed deallocation of the task's memory space [AdaLRM, 1983:13.10.1.8]. Since the Task Manager is where all tasks are created and deleted, no task returns the memory it consumes until the Task Manager terminates, and since the Task Manager never terminates, no memory from a task object is ever returned to the system. Given that a substantial number of tasks may be created during the execution of this Intelligent Real-Time system, this could lead to a significant 'memory leak'.

The term *memory leak* is a euphemism used by real-time system designers to describe a situation in which a program's memory is continually consumed without ever being replenished. The analogy to a leak in a water bucket is clear. In the case of the Task Manager, task objects never return the memory consumed, the more task objects that are declared, the more memory has 'leaked' out of the system. Eventually, the system consumes all available memory and no additional tasks can be accommodated.

5.1.3. Dynamic Task Creation and Control. To combat the memory leak problem, a method was developed that allowed tasks to be reused. The tasks managed by the Task Manager are divided into three types: periodic, singular, and any-time tasks. Each of the three task types has its own Ada task type declared for it. Each Ada task type is encapsulated in its own Ada package along with a buffer used to communicate with tasks of that type. Each of the Ada packages provides externally visible procedures that allow an external task (i.e., the Task Manager) to store items into the buffer, create new tasks of that type, and remove items from the buffer. The Ada package specification for each task type additionally exports an access type for that task type and a variables record type used to pass information from the Task Manager to a specific task or the buffer and vice versa. *The method used to do dynamic task creation, reuse, and scheduling is a fundamental accomplishment of this thesis research.* Figure 5.1 illustrates the concept and how it is implemented.

Each of the tasks P_1, S_1, A_1, etc., shown in Figure 5.1, acts more as a wrapper or shell than a stand-alone self-contained task. The task shell essentially provides the system with a reusable task that can be directed to execute any of the procedures that are contained in that task type's package. As each task type is declared or used, it is passed a parameter that specifies what procedure to use. The task type contains a case statement for invoking the correct procedure based upon the passed parameter. Note that extending this concept to include new or different task types, or additional procedures for a task type, is relatively easy should it become necessary.

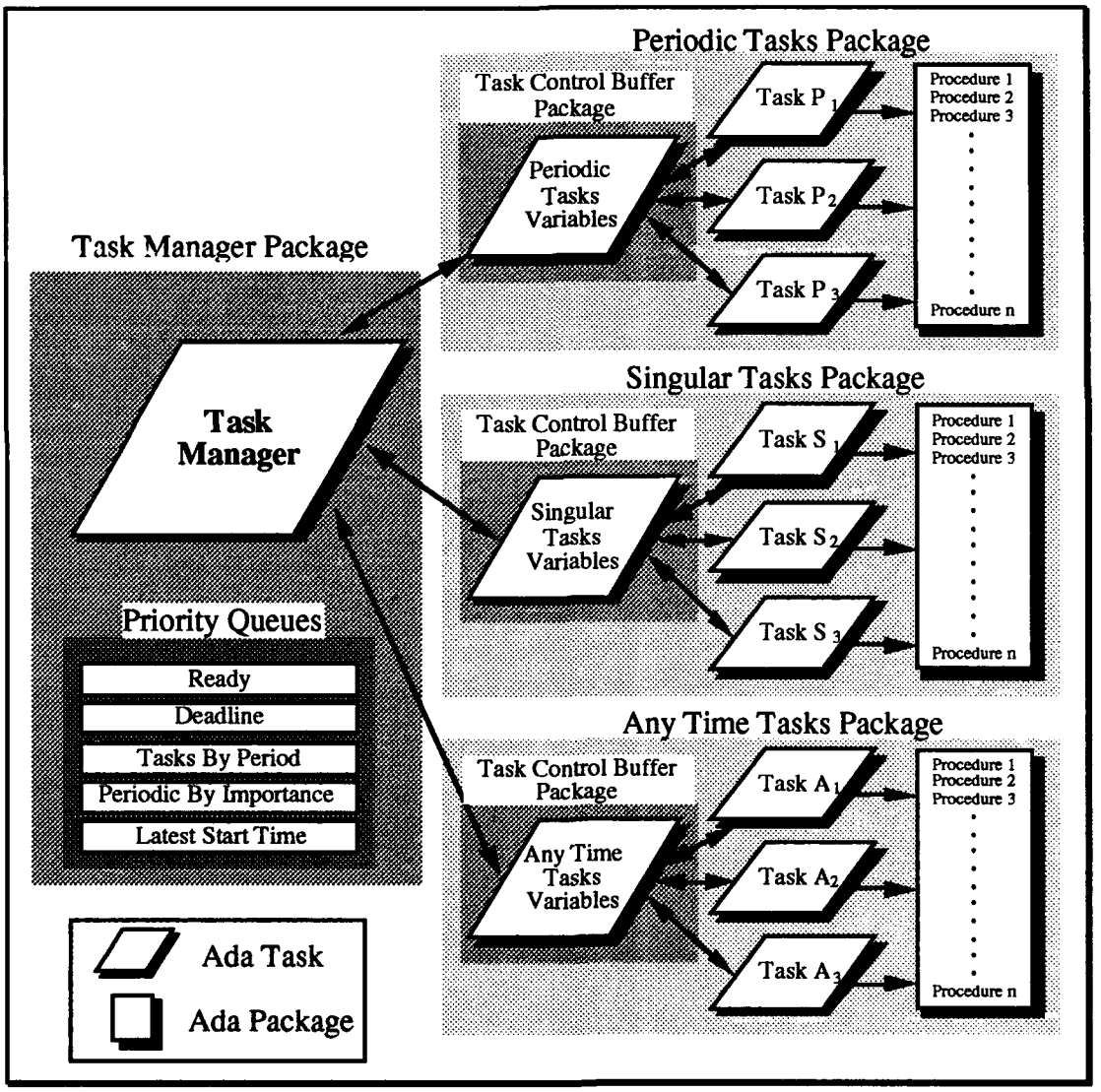

Figure 5.1 Package Structure of Tasks

Each task also must provide at least two task entries called 'Initialize' and 'Change_Variables' that accept the variables record defined for that task type. The Initialize entry is used by the externally visible procedure Create, to instantiate a new task and provide the Task Manager with information about that task. In particular, the Create/Initialize pair provides the Task Manager with the Verdix Ada defined Task_ID, and the execution times of the task. The Task_ID is needed to use the Set_Priority, Suspend_Task, and Resume_Task procedures and the durations are needed to determine the scheduling of the task.

The Change_Variables task entry is used by the Task Manager to put a previously used task 'shell' into a new known initial state. It returns to the Task Manager the new durations of the procedure associated with the task shell. Additionally, after either of these entries/procedures is called, the task suspends itself using the Verdix Ada Suspend_Task procedure. Note that all task shells not currently in use are assumed to be in a 'suspended' state by the scheduler.

The reason for using the Suspend_Task procedure may not be obvious. Ada has a delay statement that can be used to put a task to sleep until some specified duration has elapsed. The language standard guarantees that the minimum time the task sleeps can be specified by the duration given to the delay statement, but it does not specify the maximum amount of time the delay statement may consume. For real-time systems, this is an unacceptable situation. Ensuring some exact period for a periodic task means not relying upon the delay statement. Usually, a clock that generates an interrupt is used to ensure exact durations. However, in this thesis, task execution and suspension is explicitly controlled using the Verdix supplied Suspend_Task and Resume_Task procedures and a task dispatcher. The only place that a delay statement is used to ensure task timings is in the task dispatcher. This was a deliberate decision to ensure that the addition of an interrupt driven clock should be simple for any future work.

The existence of each of the Task Variables buffers provides the Task Manager an easy method to control each task. When each task starts its execution cycle, it checks in its variables buffer for control inputs from the Task Manager. The control inputs consist of a display flag, an execution mode, a procedure identifier, and an Environment Model object identifier. Any-time task types also include a boolean variable that tells the task whether or not it should continue executing, if it is executing its optional part. The display flag is a debugging aid that tells the task shell whether or not to print out its execution statistics. The execution mode is either mandatory or optional, and the procedure identifier and Environment Model object identifier tell the task shell which procedure to execute and which piece of data in the Environment Model to use.

5.1.4. Top Level Priority Assignments. The methods used to assign priorities to the different task types is built upon a number of underlying assumptions, some previously stated, others not. Before we begin the discussion of priority assignments though, the difference between priority and importance needs reiterating. The priority of a task determines the execution order of the task and the importance of a task has to do with its global relevance to the problem at hand.

The basic assumption that the system is at times overloaded which implies that at times the priority of a task must be driven by its importance. Conversely, using the rate monotonic algorithm for periodic tasks implies that when the system is not overloaded, for periodic tasks, importance is almost meaningless to the scheduler. The problem faced in the implementation then is under what conditions and how should different priority schemes be used, and what are some practical schemes to use.

Since the feasibility demonstration system is being implemented on a single processor system, the priorities must also be used to simulate concurrency of the other architecture components. This causes some problems when deciding upon the priorities to assign to the Reasoning Process, I/O Process, Environment Model, System Model, and Task Manager. If the Reasoning Process is given a priority higher than that of the periodic tasks, the assumptions used in the rate monotonic theory no longer hold. Since the behavior of the Reasoning Process is not periodic, it would not be preempted by a periodic task. The result is an inability to schedule periodic tasks to meet their deadlines. Additionally, any non-periodic task with a priority higher than the periodic tasks would result in same problem. The I/O Process on the other hand should execute at a relatively high priority to ensure rapid response to external events.

The compromise solution settled upon is shown in Figure 5.2. Of the 100 task priorities available, eighty are allotted for periodic tasks (89–10) and four are allotted for non-periodic tasks (9–6). The remaining twenty priorities are used for the I/O Process, the Task Manager, emergency tasks, non-schedulable periodic tasks, and discarded non-periodic tasks. A discarded task is one that is no longer doing useful work because it has missed its deadline. The possible task states and their meanings are discussed in detail later in this chapter.

The I/O Process is assigned the highest priority (99) and the Task Manager the second highest (98). Priorities (97–90) are reserved for emergency tasks. The allowance for emergency tasks gives the Reasoning Process the ability to ensure that a specific task runs regardless of the current workload. The Reasoning Process is assigned a static priority of five (5), to prevent it from interfering with the scheduling

and execution of both the periodic and non-periodic tasks. The impact of assigning the Reasoning Process such a relatively low priority must be examined during testing. The Environment Model, System Model, and Task Variables Buffer tasks are not assigned priorities since they basically act as passive tasks, only consuming CPU time when called. The implementation and use of priority inheritance protocols by the Verdix Ada compiler ensures that these tasks do not block high priority tasks from executing.

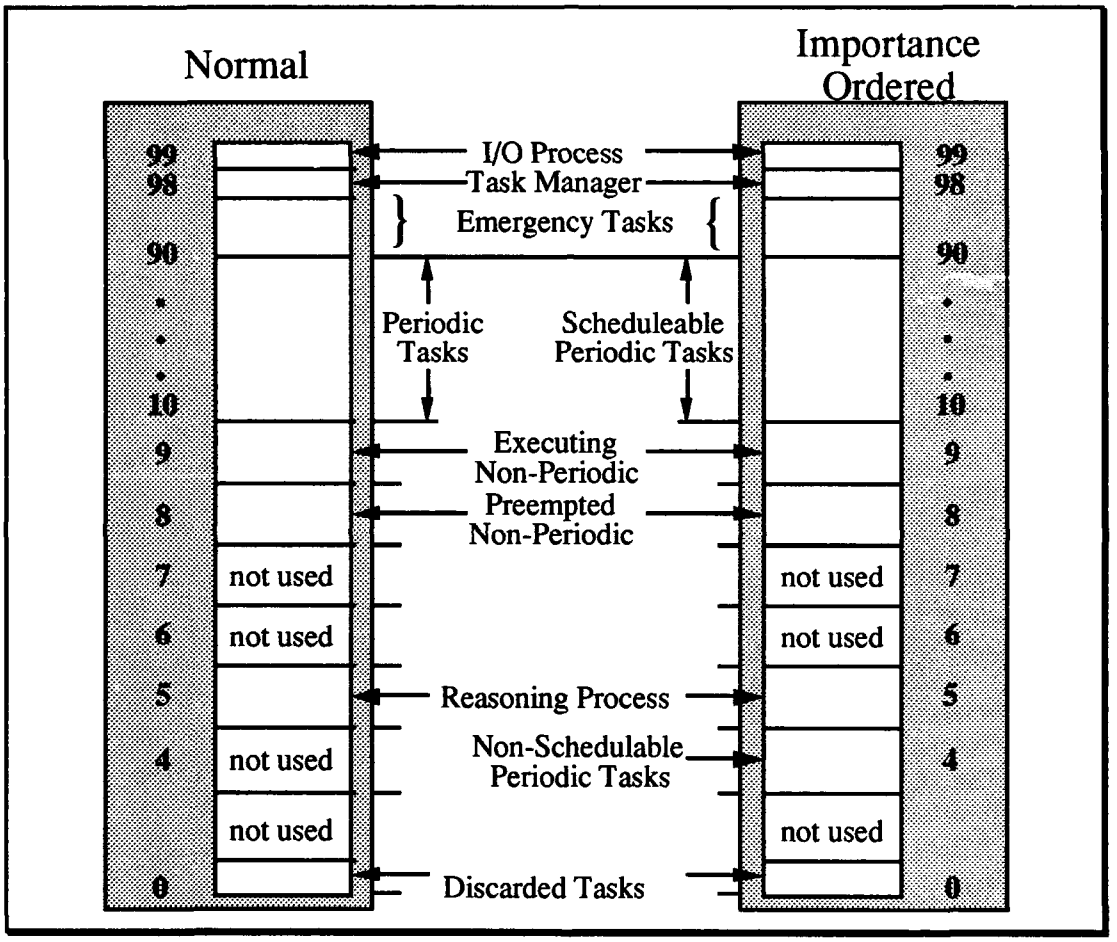

Figure 5.2 Normal and Importance Ordered Priority Ranges

The Importance Ordered side in Figure 5.2 shows the result of overload (i.e., the existence of tasks that can not be scheduled to guarantee completion by their deadlines) on the priorities assigned to the tasks in the system. The effect of overload on task priorities is primarily restricted to periodic tasks. Periodic tasks, whose importance is not high enough to put them in with the tasks scheduled rate monotonically, are assigned the bottom priority in the system. They only execute when nothing else can. Note that it may

happen that there are not enough periodic tasks to use all of the periodic priorities. In this situation, those priority values are not used.

Now that some of the general implementation issues have been addressed, the specifics of each component's implementation will be discussed. Figure 5.3 is used to facilitate the discussion. Note, however, that the focus of the feasibility demonstration is on the Task Manager and thus the other components, with the exception of the Reasoning Process, are simply acting as place holders. Their inclusion is simply to emphasize the requirement for their existence in a complete system and force the feasibility demonstration system to address them, at least in a limited way.

5.2. System and Environment Model

As mentioned previously, both the System Model and Environment Model exist only as place holders. Each of these models are implemented as Ada tasks that are passive in nature, meaning they only perform work when an entry call is made to them. These two IRTS components require significant additional work to fully implement the architecture as outlined in this thesis. However, they are not the primary focus of the feasibility demonstration and thus little effort was expended in developing them or addressing issues in their implementation.

5.3. I/O Process

For the purposes of the feasibility demonstration, the I/O process simply sends intermittent event messages to the Reasoning Process's Event_Message entry. There is no significance to the messages. They are simply used to simulate the arrival of I/O and generate an appropriate response by the Reasoning Process. For demonstration purposes, this approach suffices.

5.4. Reasoning Process Implementation

The Reasoning Process used in this feasibility demonstration is the expert system tool as implemented by CLIPS/Ada. The purpose of this discussion is not to explain the inner workings of CLIPS/Ada but rather the incorporation of it into the feasibility demonstration. Since the focus is not on the use of an expert system shell, for a detailed description of CLIPS/Ada, refer to the CLIPS user's

manuals [CLIPS-Ada, 1991] [CLIPSRefMan, 1991a] [CLIPSRefMan, 1991b] [CLIPSRefMan, 1991c] and [CLIPSUG, 1991].

The basic Ada Task structure of the Reasoning Process is shown in Figure 5.3. The Reasoning Process Ada package consists of an Ada task to encapsulate the entire process and two CLIPS/Ada defined packages, Embedded_CLIPS and User_Functions. Embedded_CLIPS is the CLIPS/Ada inference engine, modified as the name implies to be embedded in other applications instead of being a standalone application. The User_Functions package is a user-defined package that contains the interface between CLIPS/Ada and any user-defined functions.

The user-defined functions developed and coded for this feasibility demonstration allow CLIPS/Ada to add, modify, and remove tasks in addition to changing the budgeted periodic utilization. Each of these functions is callable from either the right-hand-side or left-hand-side of rules defined in the CLIPS/Ada rule format. Specific parameters needed to use these functions can be found in Appendix A and is not discussed here. What is discussed are some problems that had to be overcome to write those user defined functions.

The first problem involved data types available in CLIPS/Ada and the conversion from external data types to CLIPS/Ada data types. First, CLIPS/Ada explicitly defines what it uses for numbers as either CLIPS_Reals or CLIPS_Integers. This forces the use of Ada explicit type conversions to communicate numbers between CLIPS/Ada and external tasks. Second, a method of referring to a particular task needed to be developed since CLIPS/Ada can not use the data type for Task Control Blocks. The approach taken was to allow CLIPS/Ada to refer to tasks by their Task_ID using the Ada unchecked conversion procedure. This approach required the Task Manager to search through the currently instantiated tasks to match a Task_ID with a Task_Control_Block, potentially adding search time to any operation directed by the Reasoning Process.

The Reasoning Process task that encapsulates CLIPS/Ada provides the interfaces between the other IRTS components and CLIPS/Ada. Currently, those interfaces consist of 1) an entry to assert a fact into CLIPS/Ada (supplied as a string), 2) an entry to receive Task Manager status updates from the Task Manager, 3) an initialize entry to load the rules used by CLIPS/Ada, 4) an entry to signal when a task has missed its deadline, 5) an entry used by the I/O Process to signal an external event, and 6) an entry to signal that an infeasible task creation request has been made. The I/O Process event message entry is used by the I/O Process to insert facts into CLIPS/Ada that stimulate CLIPS/Ada to add, remove, or modify tasks.

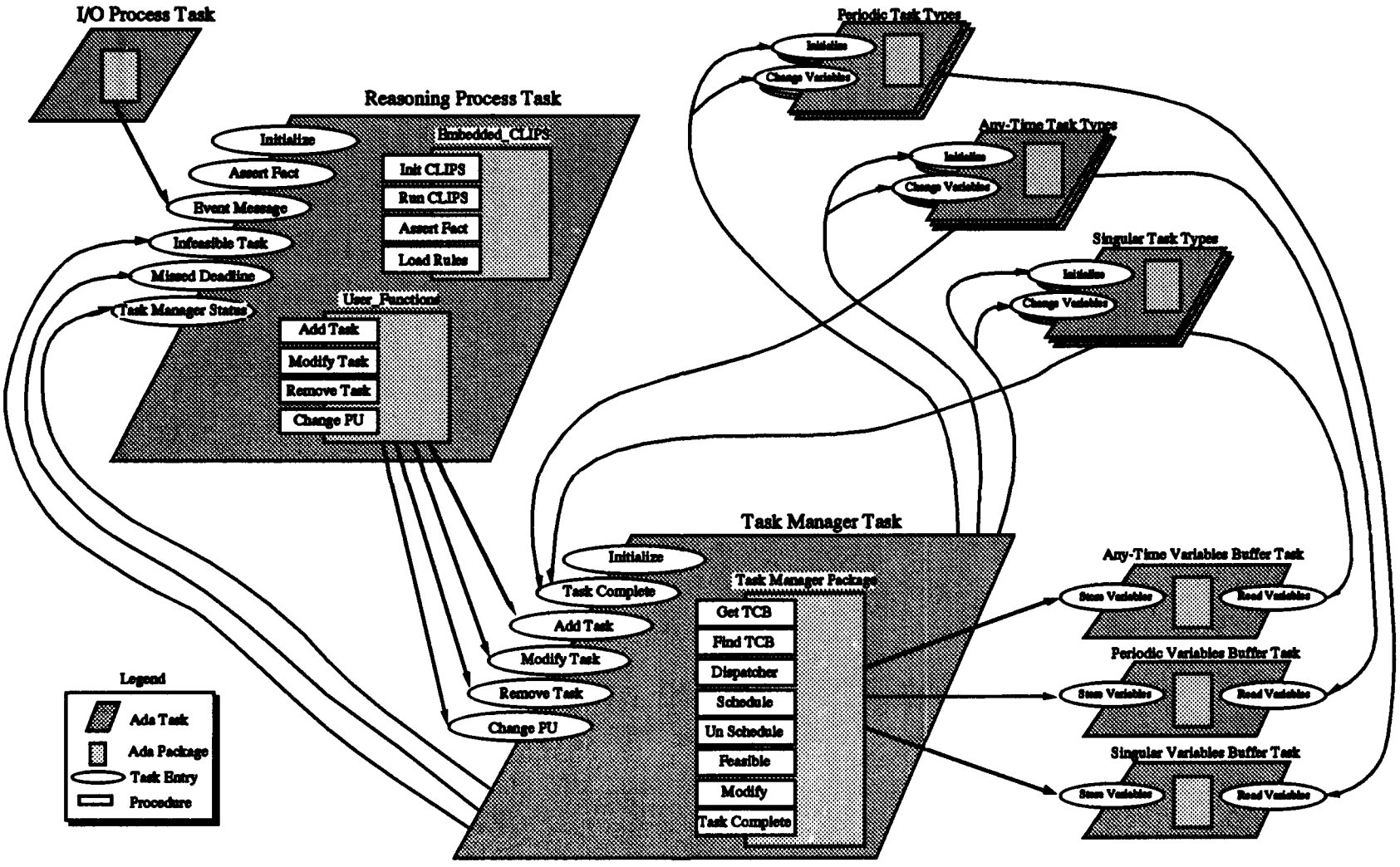

Figure 5.3. IRTS Ada Task Structure Diagram

The Reasoning Process internal task structure is arranged similarly to the system outlined in section 2.3.3. Although not implemented in the feasibility demonstration, almost all of the agenda management methods as discussed in section 2.3.3 could be easily implemented. The implementation as it stands now first checks for any entry calls to the Reasoning Process and performs any that exist. Next, one rule is allowed to fire and the cycle then repeats. Future efforts should look at fully implementing the agenda management methods developed under the Pilot's Associate program to improve and control the performance of the Reasoning Process [Dodhiawala, 1988] [Lambert, 1991] [Lambert, 1990].

5.5. Task Manager Implementation Details

In this section the Task Manager is described in detail. The description begins with the data structures used by the Task Manager. Next, periodic, singular, or any-time task state diagrams are examined along with the events that trigger a transition from one state to another. Following that, the assignment of periodic priorities is discussed. The Task Manager implementation discussion is concluded with an explanation of each major procedure it contains.

5.5.1. Task Manager Data Structures. In order to understand how the Task Manager works, it is important to outline its major data structures. The data structures consist primarily of priority deques [Booch, 1986] and a record structure called a Task Control Block. A priority deque is a queue in which items can be added or removed from either the front or the rear of the queue. In addition, the items in the queue are arranged by some user defined 'priority'. When a new item is added to the queue, it is placed in priority order; a new item with the same priority as an existing item can either be placed in front of or behind the existing item. This equal priority ordering choice allows one to implement either LIFO or FIFO ordering for items of equal priority. All items of equal priority added to a queue are added in FIFO order (i.e., behind items of equal priority).

There are six priority deques that are the most important data structures used in the Task Manager. They are as follows:

- *Ready_Queue* – A deque of tasks that are ready to execute but not yet scheduled, arranged so that the task with the earlier start times are ahead of the tasks with later start times.

- *Latest_Start_Time_Queue* – A deque of singular and any-time tasks that are currently executing, arranged such that tasks with earlier latest start times are ahead of tasks with later latest start times.

- *Deadline_Queue* – A deque of singular and any-time tasks that are currently executing, arranged such that the tasks with the earlier deadlines are ahead of tasks with later deadlines.

- *Tasks_by_Period* – A deque of only periodic tasks that are arranged with tasks of shorter periods ahead of tasks with longer periods.

- *Periodic_Importance_Queue* – A deque of only periodic tasks, arranged so that tasks with a higher importance (lower number) are ahead of tasks with lower importances.

- *Task_ID_Queue* – A deque of all tasks arranged by an integer representation of their Task ID. This deque is used to find the task control block (described below) for a given Task ID.

There is one additional priority deque that is used for scheduling periodic tasks. It is a temporary deque used when the system is overloaded. The algorithm for handling this situation involves searching through each task in the Periodic_Importance_Queue and determining which tasks can be feasibly scheduled. Those that can are added to the period-ordered temporary deque. Once all the important tasks that can be scheduled are found, the temporary deque is used to assign them priorities.

A Task Control Block is an Ada variant record type that holds the information about each task that is needed by the Task Manager. Figure 5.4 shows the Ada type declaration for the Task Control Block. There are currently three variants of the record structure, one for each of the task types (periodic, singular, and any-time). A variant record was used because each of the task types has different variables associated with its scheduling and execution, but all the tasks have some common features. The common items consist of the task's type, Task_ID, integer Task_ID, mandatory duration, optional duration, importance, priority, starting time, latest starting time, time remaining, deadline, period and status.

The different task types do not use all the common parts of a TCB in the same way. Periodic tasks use the deadline value as a stop time while non-periodic tasks have a period of zero. The variant part of a Task_Control_Block contains a pointer to the task type it is controlling and another record structure that holds the task specific variables needed by that task. Note in both cases these are data types exported by

the packaged specifications for the any-time, periodic, and singular tasks. With a basic understanding of the data structures used, the task states can be discussed.

5.5.2. Task States and State Transitions. Although there are three types of tasks (periodic, any-time, and singular), the scheduler handles the tasks as either periodic or non-periodic. All periodic tasks are scheduled rate monotonically while non-periodic tasks are scheduled with a modified earliest deadline first algorithm. This section describes the states each task type goes through and explain what events cause the transition from one state to another. For this discussion, both the any-time and singular task types are under the non-periodic task heading.

```
type Task_Control_Block_Type ( Kind : Task_Kind_Type := PERIODIC )
is record
        Deadline            : Calendar.Time ;
        Importance          : Integer ;
        Latest_Start_Time   : Calendar.Time ;
        Mandatory_Duration  : Duration ;
        Next                : Task_Control_Block_Ptr ;
        Optional_Duration   : Duration ;
        Period              : Duration := 0.0 ;
        Priority            : System.Priority ;
        Start_Time          : Calendar.Time ;
        Started_At          : Calendar.Time ;
        Status              : Status_Type ;
        Task_Kind           : Global_Data_Types.Task_Kind_Type ;
        Task_ID             : System.Task_ID ;
        Time_Remaining      : Duration ;
        Integer_Task_ID     : Integer ;
        --
        -- A case is required for every kind of task that you wish
        -- to be able to create.
        --
        case Kind is
            when Global_Data_Types.Periodic =>
                Periodic_Variables    : Periodic_Variables_Type ;
                The_Periodic_Task_Ptr : Periodic_Task_Ptr ;
            when Global_Data_Types.Any_Time =>
                Any_Time_Variables    : Any_Time_Variables_Type ;
                The_Any_Time_Task_Ptr : Any_Time_Task_Ptr ;
            when Global_Data_Types.Singular =>
                Singular_Variables    : Singular_Variables_Type ;
                The_Singular_Task_Ptr : Singular_Task_Ptr ;
       end case ;
    end record ;
```

Figure 5.4 Task Control Block Ada Record Type Declaration

5.5.2.1. Periodic Task State Transitions. A periodic task can be in one of four states, READY, EXECUTING, SUSPENDED, or COMPLETED. A periodic task in the READY state is ready to execute, but its start time has not yet been reached. An EXECUTING periodic task has passed its start time and is currently executing. A SUSPENDED periodic task is an executing task that has completed its work for that period and has called the Suspend_Task procedure. A COMPLETED periodic task has passed its deadline or more correctly, its stop time, and is now available for reuse. Figure 5.5 shows each state and labels the transition arcs from one state to another. The following discussion refers to Figure 5.5.

Event E1 occurs when a periodic task is added to the system and there are no available periodic task 'shells'. This event causes the Task Manager to use the Ada "new" command to allocate a new TCB for a periodic task. The task's variables are set and the task is inserted into the Task_ID_Queue, Ready_Queue, Tasks_By_Period_Queue, and Periodic_Importance_Queue. The task's status is then changed to READY. Note that all periodic tasks not currently executing are in the suspended state.

Event E2 occurs when the task first makes it to the front of the Ready_Queue. When the task reaches the front of the Ready_Queue and its start time equals the current time, it is started using the Resume_Task procedure and its status changed to EXECUTING. Its next start time is calculated and the task's TCB is reinserted into the Ready_Queue. When the task completes its work for the current period, it suspends itself (event E3) and is considered in the SUSPENDED state. When it again makes it to the head of the Ready_Queue, it is resumed again (event E4). The cycle continues until either the task is explicitly told to stop or its stop time has past, as signaled by event E5. It is during this cycle that periodic task missed deadlines are detected and signaled to the Reasoning Process. Again, the Resume_Task procedure detects when the resumed task was not suspended and generates an exception.

Once event E5 has occurred, the task is placed on the free task list for periodic tasks. Note that it is possible for a task to transition from the READY state to the COMPLETED state (event E5). This transition occurs when the Task Manager is told to remove a task and the task has not yet started executing. Also, since each task is in a known internal state when placed in the COMPLETED state, it can be reused without any difficulty (event E6).

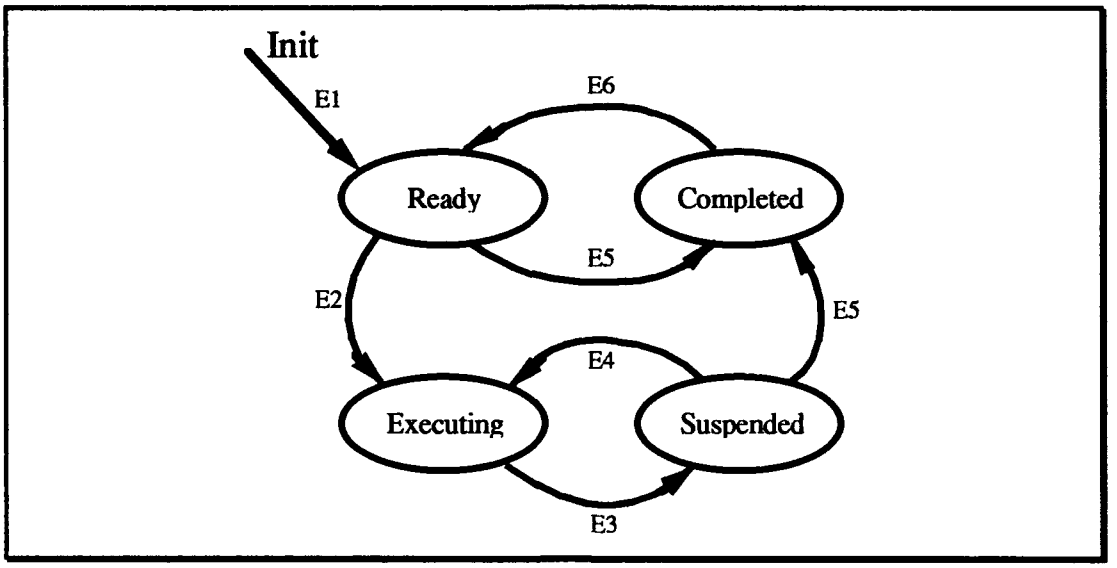

Figure 5.5 Periodic Task State Transition Diagram

5.5.2.2. Non-Periodic Task State Transitions. Non-periodic tasks have seven possible states because the scheduling of non-periodic tasks is more difficult. The non-periodic task state transition diagram is shown in Figure 5.6. In the figure, the gray lines represent the normal, non-overloaded, non-periodic task state transitions. Event E1 occurs when there are no any-time or singular task 'shells' available and one is needed. Once the new task has been allocated, its variables are set, its status changed to READY, and it is placed in both the Ready_Queue and the Task_ID_Queue. Note that a READY non-periodic task is *not* in the Latest_Start_Time_Queue or the Deadline_Queue.

From the READY state, there are two possible transitions, event E2 and event E6. Both events occur when the task reaches the head of the Ready_Queue. If the task has a deadline that is earlier than the currently executing non-periodic task, then the task is placed in the EXECUTING_MANDATORY state and is given the highest non-periodic priority (event E2). Additionally, what was the currently executing task is preempted and depending on its state, it is either placed in the PREEMPTED_MANDATORY (event E8) or PREEMPTED_OPTIONAL state (event E10). In addition, the preempted task's remaining computation time and latest start time are calculated, and the task is inserted in both the Latest_Start_Time_Queue and the Deadline_Queue. If the task coming from the Ready_Queue does not have the earliest deadline (i.e., is not at the head of the Deadline_Queue), then its state is changed to PREEMPTED_MANDATORY, its time remaining and latest start time are calculated, and it is inserted in both the Latest_Start_Time_Queue and the Deadline_Queue (event E6).

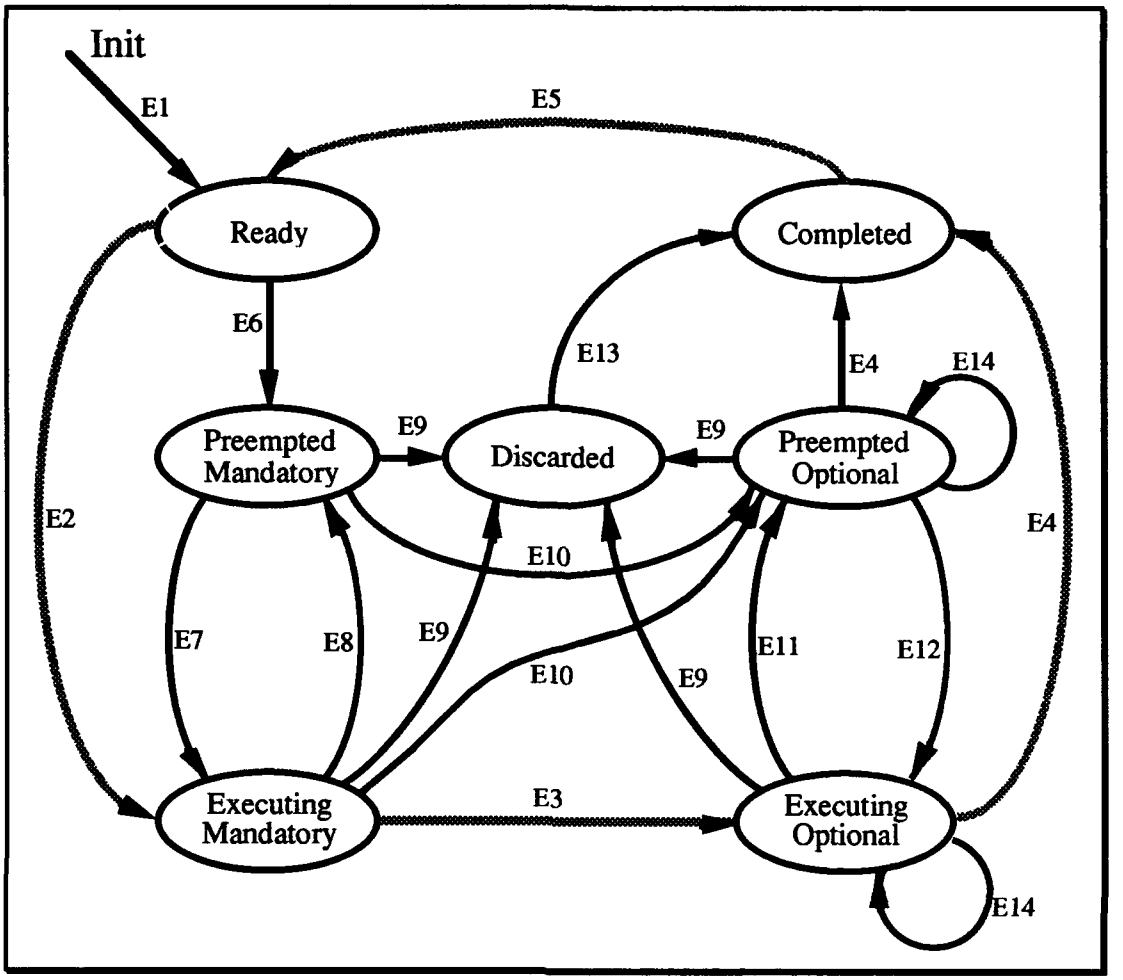

Figure 5.6 Non-Periodic Task State Transition Diagram

Note that an EXECUTING_MANDATORY or EXECUTING_OPTIONAL task is *not* in either the Latest_Start_Time_Queue or the Deadline_Queue. The reason is that in the normal case, the task has been scheduled to execute because it has the earliest deadline and, therefore, it is assumed it executes to completion. If it could not have completed before its deadline (i.e., its remaining computation time was more than the time remaining until its deadline) then a missed deadline would have been signaled for the task and the task's state changed to DISCARDED (event E9).

A task that is in the EXECUTING_MANDATORY state can either be preempted (event E8), or complete its mandatory part and be scheduled to execute its optional part (event E3), or be DISCARDED (event E9). Event E8 occurs either in the case outlined above where a newly "eligible to execute" task has an earlier deadline, or the latest start time for a task has occurred and the EXECUTING_MANDATORY

task either has slack time (i.e., its execution time remaining is less than the time remaining until its deadline) or the task at the head of the Latest_Start_Time_Queue has not completed its mandatory part (i.e., is in the PREEMPTED_MANDATORY state) and has a higher importance value than the current EXECUTING_MANDATORY task.

If the preempted task is in the EXECUTING_MANDATORY state, and does not have any slack time, then it misses its deadline (event E9). In this case, the task is placed in the DISCARDED state and its priority set to the lowest available. Note that the task cannot immediately be placed back into the free list because it is not in a known state. It must be allowed to finish its current work, but not alter either the state of shared data before it can be placed in the COMPLETED state (event E13) and in the free task list. Thus, in addition to changing its state, the preempted task's continue flag is also set to false.

If a preempted task is in the EXECUTING_OPTIONAL state, and the task does not have any slack time, then the task will miss its optional part deadline (event E9). This condition is different then missing the deadline for its mandatory part in that there is still an answer available, only the quality of the answer is affected. Thus, this case is *not* signaled to the Reasoning Process as a missed deadline. Note, however, that the task is still discarded and only marked as COMPLETED when it signals completion (event E13).

When a task that completes its mandatory part, either while in the PREEMPTED_MANDATORY or EXECUTING_MANDATORY state, new values for both its latest start time and computation time remaining is calculated. Next, the task's TCB is placed in both the Latest_Start_Time_Queue and the Deadline_Queue, and the task's state changed to PREEMPTED_OPTIONAL (event E10). If the task is at the head of the Deadline_Queue then its state will be changed to EXECUTING_OPTIONAL and it will begin executing its optional part (event E12).

The values used in the calculation of the remaining computation time and the latest start time depend upon which state the task is transitioning from. When a task transitions from the READY state to either the EXECUTING_MANDATORY or PREEMPTED_MANDATORY states (events E2 or E6), the value of the task's mandatory duration is used to determine its latest start time and the initial value of the computation time remaining. When the task transitions from either the EXECUTING_MANDATORY or PREEMPTED_MANDATORY state to the PREEMPTED_OPTIONAL state (event E10), the task's optional duration is used.

The effect of a task completing its optional part depends upon the type of task. A singular task that completes its optional part from either the EXECUTING_OPTIONAL or PREEMPTED_OPTIONAL state (event E4), is placed in the free task list and its state changed to COMPLETED (because it is possible for a task in the EXECUTING_OPTIONAL state to block for some external process, it is possible for a PREEMPTED_OPTIONAL to execute and complete). Note that the task itself will call the Suspend_Task procedure after it signals it is complete. When an any-time task completes its optional part, if there is enough time to complete another complete cycle through an optional part, its computation time and latest start time are adjusted and the task's TCB is reinserted into the Latest_Start_Time_Queue (event E14). This process is repeated until the any-time task's deadline arrives.

5.5.3. Periodic Task Priority Assignments. A number of different methods of assigning periodic priorities were investigated during the development of the feasibility demonstration. The investigation focused the problem of assigning priorities when there are more tasks than priorities. Appendix B contains a description of the various methods developed and the results of testing each method. Note that when there are fewer tasks than periodic priorities, it is a simple matter to cycle through the Tasks_By_Period_Queue to assign priorities.

The investigation concluded that for the feasibility demonstration system, the problem would be solved by simply dividing the number of periodic tasks by the number of periodic priorities. The result of that division is then used as the number of tasks to assign to each priority. Again the Tasks_By_Period_Queue is cycled through, this time assigning the appropriate number of tasks the same priority before moving on to the next priority.

5.5.4. Task Manager Entry Call Descriptions. The Task Manager is implemented as an Ada task with five entry calls. The entry calls are Add_Task, Modify_Task, Remove_Task, Change_Periodic_Utilization, and Task_Complete. The task entries Add_Task, Modify_Task, Remove_Task, and Change_Periodic_Utilization are used by the Reasoning Process to control the current task set. The Task_Complete entry is used by the any-time and singular tasks to signal the Task Manager that they have completed executing. In addition, the Ada task is set up with a delay statement that essentially places the Task Manager in an idle state when it is not in use. The following sections describe the processing that takes place whenever a call is made to any of the task entries and explains how the Ada select statement with a delay alternative is used in the Task Manager.

5.5.4.1. Add_Task Entry Call. Task creation is done in response to a request from the Reasoning Process to instantiate a new task. The process varies depending upon the type of task that requires creation, but the basic process is: getting a Task Control Block, filling in the appropriate values, determining the feasibility of the task, adding the new task to the appropriate priority queue, scheduling the task, and finally sending the control variables to the task's control buffer.

Getting a new Task Control Block (TCB) is accomplished by the procedure Get_TCB. Because the system reuses task shells it first checks the free task list to see if there are any tasks of the required type available for use. If there are, then the TCB for an unused task shell is modified to reflect the new parameters as passed to the Task Manager. Since task starting and stopping is explicitly controlled by the Task Manager, any task not currently in use (i.e., stored in the free task list) is in the suspended state. Therefore, the Resume_Task procedure is called and a call to the task's Change_Variables entry is made. The call to the task's Change_Variables entry returns to the Task_Manager the task's durations and the values are stored in the task's TCB. If there are no task shells of the right type available on the free task list, then a new task of the appropriate type is instantiated and a similar call to the Change_Variables entry is made. Note the task itself has again been placed in the suspended state after it responds to the Change_Variables entry call.

Once the task's durations are returned to the Task Manager, a call to test the feasibility of the new task is made. The feasibility check for a periodic task consists of determining if the task's period is greater than the task's combined mandatory and optional durations and if at least its mandatory part can be executed without exceeding the current budgeted periodic utilization. The feasibility check for a non-periodic task consists of determining whether the task can complete its mandatory part before its deadline. If the task is feasible, then it is placed into the Ready_Queue and the Reasoning Process is notified of the new status of the task set.

Deciding what to do with a new task that is infeasible is an area where future work is required. The chosen method is to make a distinction between periodic and non-periodic tasks. If a periodic task is infeasible (i.e., the task set, which includes the new task, utilization exceeds the amount available), it is added to the Ready_Queue anyway. The assumption is that at some future time there may be enough processor time available to execute the task. Non-periodic infeasible tasks are simply removed from the system altogether. The assumption made here is that the deadline is going to be missed, so the Reasoning Process has to adjust its request.

If the new task is a periodic task, then the parameters used for assigning periodic priorities are updated. In particular, if there are more periodic tasks than periodic priorities, then the ranges of periods assigned to each priority is updated. If not, then there is no need to perform this operation and some time is saved by not doing it. The periodic task's TCB is then added to the Ready_Queue, Task_By_Period, and the Periodic_Importance_Queue. If the new task is not periodic, then the number of active non-periodic tasks is incremented by one and the task's TCB is added to the Ready_Queue.

New periodic tasks are also assigned an execution priority when they are added to the system. In addition, periodic tasks are assigned a new execution priority whenever a periodic task is modified, removed, or the budgeted periodic utilization is changed. The steps in assigning periodic priorities are as follows:

1) If the current periodic utilization is below the utilization required for running all tasks with their optional parts, then instruct each task to execute both its mandatory and optional parts. Assign each task's priority rate monotonically. This condition is called ALL_OPTIONAL in the program.

2) If the current periodic utilization is above the utilization required for running all tasks with their optional parts, but below that required for all tasks to execute only their mandatory parts, then assign all tasks to execute their mandatory parts and assign each task's priority rate monotonically. Then loop through the Periodic_Importance_Queue adding optional parts until the utilization equals the required utilization. This condition is known as SOME_OPTIONAL in the program.

3) If the current periodic utilization is above the utilization required for running all tasks with their mandatory parts, then loop through the PERIODIC_IMPORTANCE_QUEUE adding mandatory parts until the utilization equals the required utilization. Then assign this subset of tasks to execute their mandatory parts and assign each task's priority rate monotonically. This condition is known as SOME_MANDATORY in the program.

5.5.4.2. Modify_Task Entry Call. A task can only be modified in a limited number of ways. Only the task's period, importance, start-time, deadline, or display flag can be changed and only when the task is in a certain state. Changing the task's procedure identifier is treated as adding another task and is not allowed as an external modifying option by the current implementation. With the exception of

changing the display flag, all the changes may involve rescheduling most of the tasks in the system. Table 5.1 and Table 5.2 show the allowed modify operations for each task type and the potential effects of that modification. However, the first operation is always finding the task's TCB in the Task_ID_Queue.

Table 5.1 Effects of Modifying Tasks and Periodic Utilization

Changed Item	Task Variables Buffer	Reschedule Periodics	Reschedule Non-Periodics
Display Flag	√		
Deadline	√		√
Importance	√	√	√
Period	√	√	√
Start Time	√		√
Periodic Utilization	√	√	√

Modifying a periodic task's period or importance can have time-expensive consequences. It is possible for the modification to require communicating a significant number of changes to the periodic task's variables buffer. For example, changing a task's period enough to change the periodic condition from ALL_OPTIONAL to SOME_MANDATORY will require each periodic task's variables to be changed. Additionally, since the Display Flag is specific to each task, and changes must be signaled to each task, it also requires the task's variables record stored in the task's variables record buffer be changed.

Modifying a non-periodic task is not quite as expensive. The worst case occurs whenever the task that is modified has started its execution. In this case, the task is modified and then a reevaluation of which task to execute is made. This reevaluation is made by checking both the Latest_Start_Time_Queue and the Deadline_Queue and executing the appropriate task.

Table 5.2 Allowed Modify Operations by Task Type and State

Task Type	State	Allowed to Modify	Actions Taken
PERIODIC	READY	PERIOD	Recalculate the Utilizations Adjust the Tasks by Period Queue Reassign the Periodic Priorities
		START_TIME	Adjust the Start Time Ordered Queue
		DEADLINE	No Action Required
		IMPORTANCE	Adjust the Importance Ordered Queue
	EXECUTING	PERIOD	Recalculate the Utilizations Adjust the Tasks by Period Queue Reassign the Periodic Priorities
		DEADLINE	No Action Required
		IMPORTANCE	Adjust the Importance Ordered Queue Recalculate the Utilizations Reassign the Periodic Priorities
	COMPLETED	NONE	Error Condition
ANY-TIME SINGULAR	READY	DEADLINE	No Action Required
		IMPORTANCE	No Action Required
		START_TIME	Adjust the Start Time Ordered Queue
	EXECUTING_MAN	DEADLINE	Stop Current Task
		IMPORTANCE	Place in LST and Deadline Queues
	EXECUTING_OPT	DEADLINE	Pick new Current Task
		IMPORTANCE	
	PREEMPTED_MAN	DEADLINE	
		IMPORTANCE	
	PREEMPTED_OPT	DEADLINE	
		IMPORTANCE	
	DISCARDED	None	Error Condition
	COMPLETED	None	Error Condition

5.5.4.3. Remove_Task Entry Call. The actions taken when a task is removed depend upon the task type and the task's current state. In any case, however, the Task_ID_Queue must first be searched to locate the task's TCB. For a periodic task, the removal process is as follows:

1) Remove the task from the Tasks_by_Period, Periodic_Importance_Queue, and Ready_Queue.

2) Adjust the values of the mandatory utilization, optional utilization, and required utilization based upon the task's current execution mode.

3) Change the task's state to COMPLETED and place the TCB on the free task list.

4) Reschedule the remaining periodic and non-periodic tasks.

Note that there is no communication with the periodic task itself. This is possible because each task suspends itself after it has performed its work and, in the worst case, the task terminates after the current execution cycle it is performing. Allowing the periodic task to complete its cycle also ensures that the task, and the data it is manipulating are in known states.

Removing a non-periodic task is only slightly more complicated, mainly because direct communication with the task may be necessary. The procedure is as follows:

1) If the task's state is READY, then it has not started executing yet, so:

   a) Remove the task from the Ready_Queue.

   b) Change the task's state to COMPLETED and place the TCB on the free task list.

2) If the task's state is EXECUTING_MANDATORY, EXECUTING_OPTIONAL, PREEMPTED_OPTIONAL, or PREEMPTED_MANDATORY, then it has started executing, so:

   a) If the task's state is PREEMPTED_MANDATORY or PREEMPTED_OPTIONAL, then remove the task from the Deadline_Queue and Latest_Start_Time_Queue. Otherwise, the task is not in either queue.

b) Change the task's continue flag to FALSE and send the new value to the task's variables buffer. Also change the task's state to DISCARDED.

c) If the task's state was EXECUTING_MANDATORY or EXECUTING_OPTIONAL, then make the task at the head of the Deadline_Queue the currently executing task.

5.5.4.4. Change_Periodic_Utilization Entry Call. Because the budgeted periodic utilization value is used to schedule both periodic and non-periodic tasks, changing the budgeted periodic utilization can be the most expensive entry call. Whenever the budgeted periodic utilization is changed, all periodic tasks priorities are reassigned. This may require communicating execution mode changes to a large number of periodic tasks.

Non-periodic tasks use the budgeted periodic utilization when calculating their execution times. Once the new budgeted periodic utilization is set, the latest start time and time remaining for every non-periodic task is recalculated. However, since the effect is the same on all non-periodic tasks (i.e., they are each changed by the same factor) the current scheduling method is continued.

5.5.4.5. Task_Complete Entry Call. The Task_Complete entry call is used by the non-periodic tasks to signal the Task Manager that they have completed execution of their assigned part (mandatory or optional). Again, the effect of the entry call depends upon the status of the task that calls the entry. The effect of the call for the different task state's are as follows:

1) If the calling task's state is DISCARDED, then the task's TCB is placed on the free task list and no other action is required.

2) If the calling task's state is EXECUTING_MANDATORY or PREEMPTED_MANDATORY, then the task's latest start time and time remaining is recalculated using the task's optional duration and the task is reinserted into the Latest_Start_Time_Queue and Deadline_Queue. The task's state is changed to PREEMPTED_OPTIONAL. If the task's state was EXECUTING_MANDATORY then a new non-periodic task is executed.

3) If the calling task's state is EXECUTING_OPTIONAL or PREEMPTED_OPTIONAL and the task is a singular task, then the task's state is changed to COMPLETED and its TCB is placed

on the free task list. If the task's state was EXECUTING_OPTIONAL, then a new non-periodic task is executed.

5.5.4.6. Task Dispatcher. The goal of the Task Dispatcher is to implement the schedule. It does this by starting tasks from the Ready_Queue. Tasks are started using the Resume_Task procedure. Additionally, the dispatcher will delay the Task Manager until the next scheduled action is to occur. This delay statement ensures that other tasks are given processor time if needed. The dispatcher removes tasks from the Ready_Queue and operates on them as follows:

1) If the task is a periodic task then:

   a) If its stop time has not been exceeded, the task is resumed, its next start time calculated, and it is placed back on the Ready_Queue.

   b) If its stop time has been exceeded, the task's status is changed to COMPLETED and the task's TCB is placed on the free task list.

2) If the task is non-periodic then:

   a) If there is no currently executing task, set the task's status to EXECUTING_MANDATORY and resume the task.

   b) If there is a currently executing task, and the new task's deadline is before the currently executing task's, preempt the currently executing task, set the new task's state to EXECUTING_MANDATORY and resume the new task.

   c) If there is a currently executing task, and the new task's deadline is after the currently executing task's, calculate the new task's latest start time and time remaining, place the task in the Latest_Start_Time_Queue and Deadline_Queue, and change the new task's state to PREEMPTED_MANDATORY.

Once all the tasks in the Ready_Queue that need to be started have been started, the time until the next scheduling event is calculated. This time is the lesser of the first time in the Latest_Start_Time_Queue, Deadline_Queue, or Ready_Queue. The delay is then used to 'sleep' the Task Manager until that delay expires or an entry call to the Task Manager is made.

5.6. *Implementation Summary*

This chapter has spelled out the important details of the feasibility demonstration system. It has discussed the compiler choice, memory management issues, task states, priority assignments, and scheduling. The implementation handles both periodic and non-periodic tasks, detects missed deadlines and overload situations, and responds to those conditions as required. In addition, it provides for the dynamic creation and control of Ada tasks. These mechanisms provide "hooks" needed by the Reasoning Process to specify the current task set and influence its run-time scheduling. These capabilities are crucial to managing dynamic, real-time, periodic and non-periodic task scheduling.

The following chapter presents the results of testing done to confirm the operation of the developed system and thus, the feasibility of the developed architecture. The code developed for the demonstration system is included in Appendix A.

VI. Results and Analysis

Since accurate timing analysis is both impractical and unwarranted at this stage (because the design is implemented upon a multi-user UNIX platform), the approach used to demonstrate feasibility is more an existence proof of desired capability. In particular, the architecture includes, and this investigation is focused on, a Task Manager that dynamically creates, schedules, and executes the real-time tasks of an IRTS. Additionally, the architecture specifies that the Task Manager receives commands from the Reasoning Process dictating which tasks are to be created, modified, or removed. In addition, the Task Manager accepts some control inputs about how they should be scheduled. The following sections discuss the results of capability tests beginning with an assessment of the overall architecture feasibility. Next, the system's ability to dynamically create and control Ada tasks is addressed. Finally, the particular scheduling policies discussed in previous chapters are examined.

6.1. Architecture Feasibility

The question to be addressed is, "Does the feasibility demonstration lend one to believe that the architecture as outlined in Chapter 4 is viable?" The answer, from testing of the system's performance to date is, "yes, with some modifications". In particular, tests indicate that the current CLIPS/Ada based Reasoning Process executes too slowly. The reasons for the inadequate performance of the Reasoning Process have not been fully explored, primarily because the majority of the effort was in developing the Task Manager component of the architecture.

Figure 6.1 and Figure 6.2 show examples of the Task Manager overhead incurred by periodic and non-periodic tasks of different durations. The graphs assume each task uses an existing task shell when added, is modified once, and then removed. Figure 6.1 assumes an add task operation takes 0.004 seconds, a modify operation takes 0.001 seconds, and a remove operation takes 0.001 seconds. Similarly, Figure 6.2 assumes an add task operation takes 0.004 seconds, a modify operation takes 0.003 seconds, and a remove operation takes 0.001 seconds. The times are derived from testing results contained in Appendix A. Assuming a desired maximum overhead of 10%, the results indicate the approach is feasible for systems whose task durations are about 0.07 seconds or greater. However, two observations are in order.

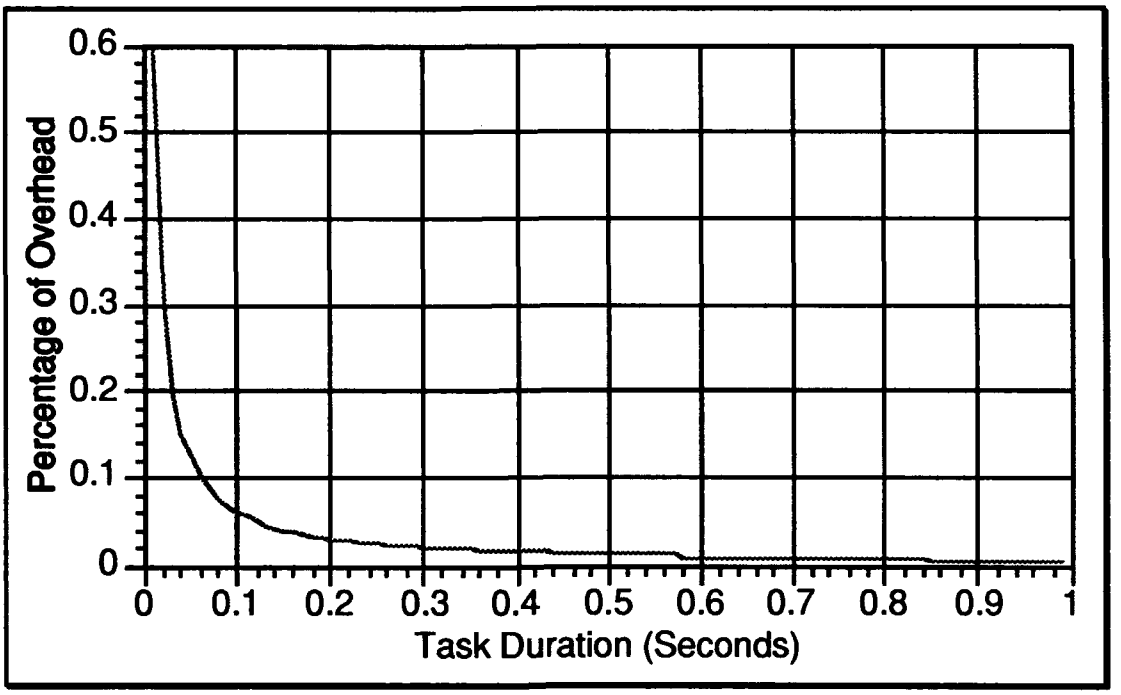

Figure 6.1 Example Periodic Task Manager Overhead versus Task Duration

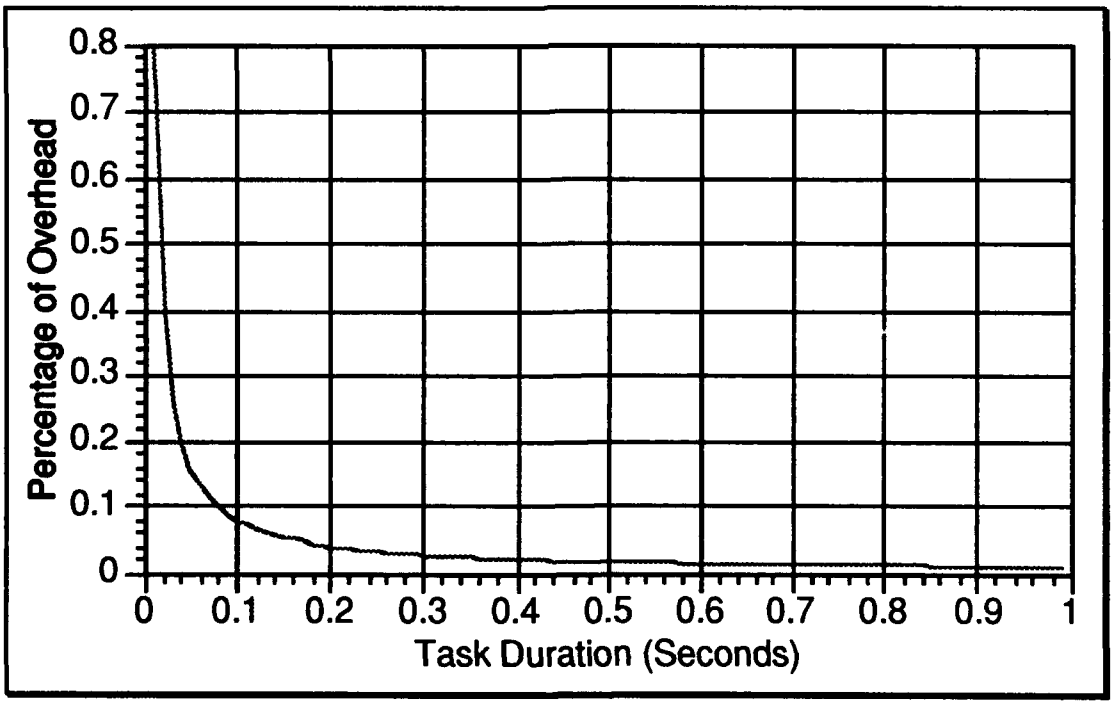

Figure 6.2 Example Non-Periodic Task Manager Overhead versus Task Duration

First, because the feasibility demonstration is currently executing on a single processor machine, and there are numerous higher priority tasks executing at any one time (the Reasoning Process priority is 5, periodic priorities start at 10 and go to 90), the reasoning process, as implemented, does not receive adequate processor time. Two conclusions can be drawn. One is the need for at least one additional processor in the system, and the other is a different priority for the Reasoning Process. Raising the priority of the Reasoning process in the current implementation is not a practical solution because the execution time of the Reasoning Process is unpredictable. This means task deadlines could not be guaranteed if the tasks had priorities below the Reasoning Process's. The better solution is a separate processor for the Reasoning Process. Most IRTSs examined in the background research required multiple processors and this feasibility demonstration has simply reiterated that requirement.

Second, because of the execution speed difference between the Reasoning Process and the Task Manager, a "message buffer" should be used whenever the Task Manager communicates with the Reasoning Process. As implemented currently, whenever the Task Manager wishes to communicate information to the Reasoning Process, it makes an entry call directly to the Reasoning Process. If the Reasoning Process is not ready to accept the entry call, the Task Manager will block. The inclusion of the buffer would allow the Task Manager to complete its control actions independent of the Reasoning Process. In addition, the Reasoning process could exert more control over its I/O.

6.2. Dynamic Task Creation and Control

A major accomplishment of this thesis is the development of an ability to dynamically create and control Ada tasks. One of the fundamental assumptions of this thesis is the existence of a plan/goal graph or task network for the chosen domain. The mapping of that problem domain structure into a solution requires the ability to dynamically create and control real-time tasks. The results of testing the feasibility demonstration indicate that the methods outlined in Chapter 5 for just that purpose are feasible.

Figure 6.3 and Figure 6.4 summarize the testing results contained in Appendix A. Each graph shows the minimum time required to: 1) adding a task that requires instantiating a new task shell, 2) adding a task that reuses a task shell, 3) modifying a task, and 4) removing a task. The effect of UNIX multiprocessing operating system is evident in the fluctuating minimum times. The graphs indicate the design approach is feasible for applications whose real-time response requirements are on the order of milliseconds. However, testing indicated a potential problem area.

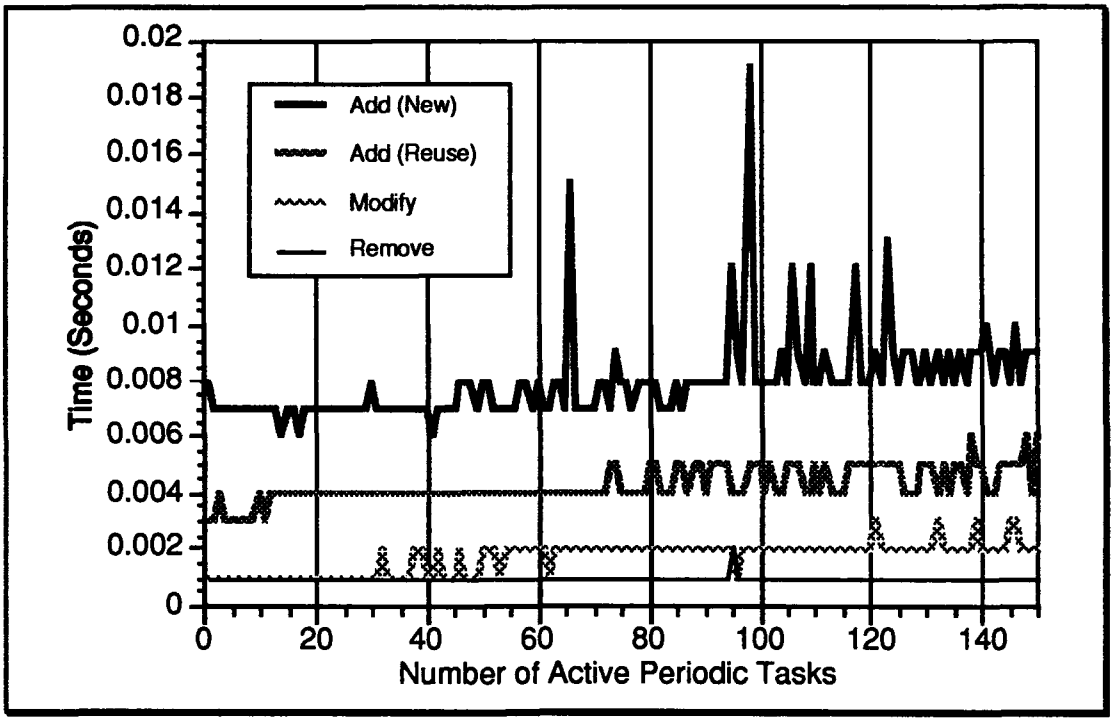

Figure 6.3 Summary of Periodic Task Control Times

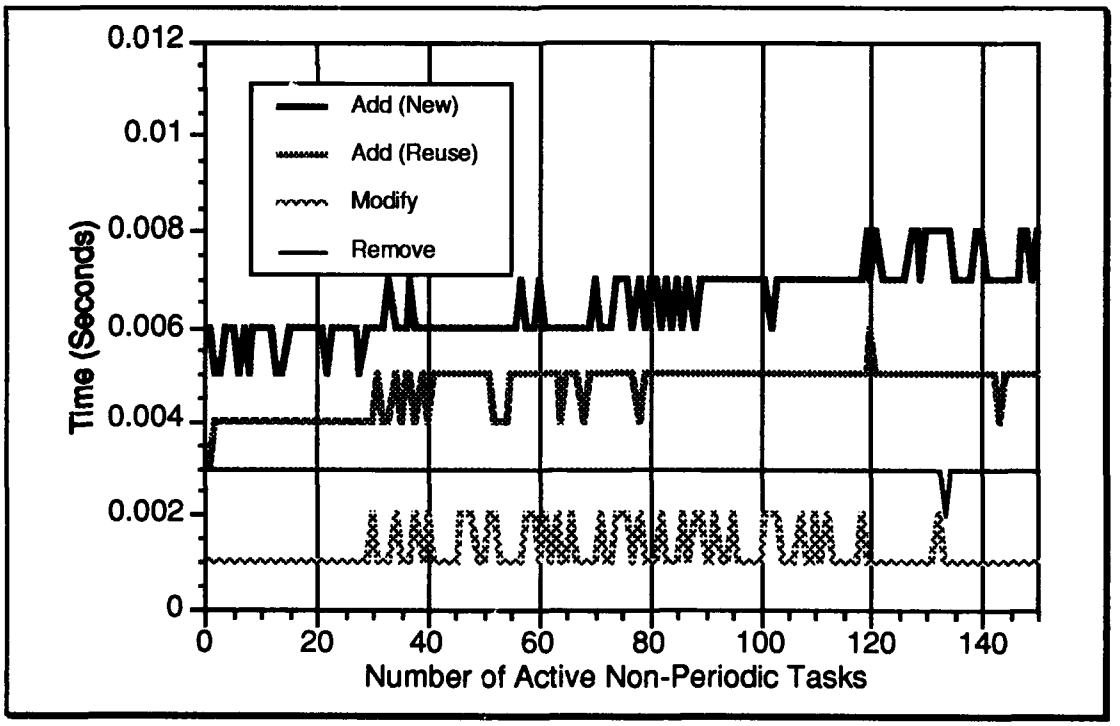

Figure 6.4 Summary of Non-Periodic Task Control Times

6.2.1. Dynamic Task Creation Results. Test results show the time required to add and schedule either a periodic or non-periodic task is around 0.004 seconds. Graphs of the results are included in Appendix A. The results can be interpreted to mean that the minimum response time for the feasibility demonstration is 0.004 seconds. The time to add a task is not the same for reusing a task or instantiating a new one.

The varying times to add a new task to the system is a problem. Testing results indicate that the average time to add a task that reuses a task shell differs from the time to instantiate a new task shell by about 0.004 seconds. The results were obtained by making numerous runs which create all new tasks, and reuse tasks and averaging together the fastest ten times from each run. The key here is not the exact times (as mentioned previously, exact timing numbers are subject to uncontrolled error), but rather the variance between the two sets of times.

The reason for the variance is the extra overhead required to instantiate a new task, however, no provisions have been made in the implementation to address it. What this means is that the Reasoning Process does not currently have a method to determine how long it will take in terms of overhead to add a new task. One solution to the problem is to provide the Reasoning Process with a running count of the instantiated but unused task shells. Using this information, the Reasoning Process could adjust its calculations of start times and execution times to account for the additional overhead if a new task is required.

A second, and perhaps better approach, is to instantiate a predetermined number of each type of task shell. This approach would allow a number of improvements. First, the variance in task creation times would be eliminated. Second, it becomes easier to detect and handle potential memory shortages. If the number of instantiated tasks is assumed to be the maximum allowed, an attempt to add a new task that would exceed that maximum is easily detected. Once detected, it is then possible to determine which task, if any, should be abandoned in favor of the new task.

6.3. Task Scheduling Evaluation

The evaluation of scheduling performance has to be looked at only to detect major deficiencies. From that point of view, there were two significant problems encountered and one interesting anomaly. The first problem dealt with the way missed deadlines were handled, and the second problem dealt with the

way durations for non-periodic tasks were calculated. The interesting anomaly dealt with the ordering of periodic tasks with the same periods.

Originally, when a deadline violation was detected, a message was sent to the Reasoning Process signaling the violation. In the case of a non-periodic task, the deadline violation was detected when the task's TCB made it to the head of the Latest_Start_Time_Queue, and the task dispatcher discarded the task. For periodic tasks, a missed deadline was detected when the task was instructed to resume for its next execution period. For non-periodic tasks, it was reasoned that the Reasoning Process would determine whether or not to extend the deadline, cancel the task, or take some other action. The same was thought to be the case for periodic tasks. For periodic tasks, this approach turned out to be infeasible.

The problem stemmed from the method used to insure the periodicity of the periodic tasks; adding the task's period to the last start time rather than the time the task is told to execute. What happens is a cascading of missed deadlines because the time used to figure the next starting time of the periodic task does not take into account any previous missed deadlines. In effect, it still tries to meet the deadline of the next cycle even if it too has already past. The correction for this problem was to restart the period from the time the missed deadline was detected, thus "forgetting" the cycle(s) that missed its deadline.

The problem with non-periodic task durations stems from the method used to adjust the duration of a non-periodic task to account for the processor time consumed by the periodic tasks. The current method uses the formula:

$$\frac{m_i}{(1-BU)} \quad or \quad \frac{o_i}{(1-BU)}$$

where m_i is the mandatory duration of the task, o_i is the optional duration, and BU is the current budgeted periodic utilization. Because the budgeted periodic utilization is an average value, and not an accurate reflection of a particular duration, as long as the non-periodic tasks are of relatively long durations, the formula works well. However, it quickly fails to provide accurate predictions when non-periodic durations are short.

One possible method to increase the accuracy of the predicted non-periodic durations is to create a periodic task that is used to execute non-periodic tasks. This method would allow the Reasoning Process to guarantee some minimum response times. By adjusting the duration and period of the task, the amount of

processor time allotted to non-periodic tasks could be better regulated. The new equation for calculating the actual durations of non-periodic tasks becomes:

$$E_{t_m} = \left\lceil \frac{m}{c} \right\rceil T \quad and \quad E_{t_o} = \left\lceil \frac{o}{c} \right\rceil T$$

where T is the period of the task, c is the duration of the task, m and o are the mandatory and optional durations of the non-periodic task respectively, and E_{t_m} , E_{t_o} are the predicted execution times of the non-periodic task.

The anomaly dealing with tasks of the same period is illustrated in Figure 6.5. Because the ordering in the queues is only based upon one field in the task's TCB, there are numerous cases where a "priority inversion" can occur. This particular problem surfaced when a test case was run with all periods equal and randomly generated importances. From Figure 6.5 it is clear that the priority assigned to each task will depend solely upon its position in the queue, and not be influenced by its importance. The impact of this priority inversion, however, is not clearly understood.

In general, for periodic tasks, the importance of a task only assures that the task will be in the schedulable set, and the rate monotonic algorithm determines the priority of each task in that set. In the case of all or some tasks in the schedulable set having the same period, priorities are assigned currently first come first served. However, the set is still classified as schedulable and thus the priority assigned should not matter. The testing conducted to date was unable to determine if the anomaly could cause problems. Given better timing analysis tools and a more controlled execution environment, an understanding of the impact of this inversion could be conducted. However, no timing faults could be traced to the inversion.

6.4. Code Complexity Analysis

In Chapter 3, improving algorithm efficacy was presented as a method of obtaining real-time performance. This section examines the code that makes up the feasibility demonstration system in terms of its time complexity. The analysis presented here should assist follow-on efforts in tuning the algorithms used to obtain maximum performance from the system.

The results of the complexity analysis are shown in Table 6.1 and Table 6.2. In the tables, n is the number of currently active periodic tasks and m is the number of currently active non-periodic tasks. The

were no unexpected results; however, there are a few areas that could be improved upon. First, the number of context switches is excessive in some places and second, implementation of the task buffers needs a reexamination. The task control buffers are the primary area where a simple change in data structures can lead to significant performance gains.

Both Table 6.1 and Table 6.2 contain a column for the number of context switches. Context switches are important because they consume significant amounts of processor time, dwarfing the code complexity numbers for all but very large values of n (the average time for a context switch on the Sparcstation is about 0.00025 seconds). A context switch occurs whenever one task calls another (as is the case with the Task Manager writing to the task control buffers). The code developed did not explicitly try to reduce the number of context switches and this issue should be addressed in the future.

The task control buffers, as written, used a linked list for implementation convenience. A task that wishes to get something from the buffer provides an index value. The task buffer then searches through the link list buffer until it finds the index and returns that item to the calling task. The effect of this implementation is an $O(n)$ operation every time the buffer is accessed. Since every active task accesses the buffer at least once during its execution, this is a very inefficient data structure.

Also, there is very little structure to the access sequences. For example, the periodic task control buffer access pattern is determined by the period of the periodic tasks. If a periodic task terminates and then is reused, its period is most likely to be different than the previous one. In addition, the set of items in the buffer only changes when a new task is *instantiated*, not when a task shell is reused.

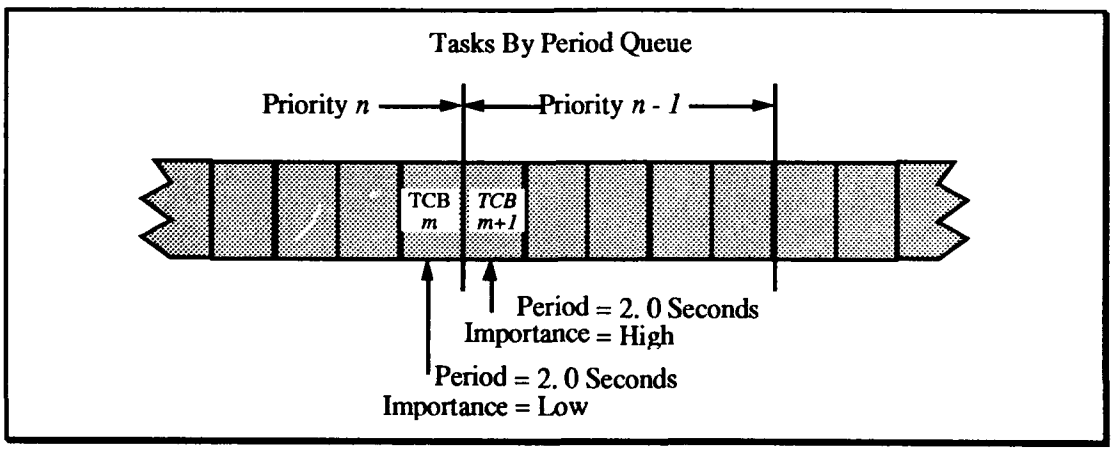

Figure 6.5 Example of Execution Priority Inversion

Table 6.1 Time Complexity of Procedures Used By the Task Manager

Procedure Name	Context Switches	Complexity for Periodic Tasks	Complexity for Non-Periodic Tasks
Assign_Periodic_Priorities	3n	$O(n^2)$	
Dispatch Tasks	SA*	O(n * SA)	O(m * SA)
Feasible		O(1)	O(1)
Find_TCB		O(n + m)	O(n + m)
Get_TCB	2	O(3n + m)	O(n + m)
Modify	3n**	$O(n^2)$	O(m)
Non_Periodic_Completed	1		O(m)
Periodic_Priorites_By_Importance	n	$O(n^2)$	
P_Tasks_<_P_Priorities	2n	$O(n^2)$	
P_Tasks_>_P_Priorities	2n	$O(n^2)$	
Schedule	1	O(n + m)	O(n + m)
Some_Periodics_Optional	n	$O(n^2)$	
Un_Schedule	1***	O(n)	O(m)

* SA stands for scheduling action, thus the order depends upon how many tasks need to start or stop at the time the procedure is called.

** only one context switch required for non-periodic task modify

*** only the non-periodic tasks require a context switch

Table 6.2 Time Complexity of Task Manager Entry Calls

Task Entry Name	Procedures Called	Context Switches	Complexity for Periodic Tasks	Complexity for Non-Periodic Tasks
Add_Task	Get_TCB, Feasible, Schedule, Un_Schedule	3n + 1	$O(n^2)$	O(n + m)
Modify_Task	Find_TCB, Modify, Feasible	3n + 1	$O(n^2)$	O(m)
Remove_Task	Find_TCB, Un_Schedule	1	O(n)	O(m)
Change_BU*	Assign_Periodic_Priorities	3n + 1	$O(n^2)$	O(m)
Task_Complete	Non_Periodic_Completed	1		O(m)

* Changing the budgeted periodic utilization affects both the periodic and non-periodic tasks.

Given the random access patterns and the fairly stable number of buffer items, a balanced binary tree data structure may be much more efficient. Since the search through a balanced binary tree takes O(log *n*) time, the speed of the buffer access could be significantly enhanced. The increased efficiency of this heavily used data structure should provide substantial performance gains. Note that the Tasks_By_ID_Queue could also benefit from this approach for the same reasons.

Although testing was not specifically conducted on the use of the priority deques, their use appears extremely beneficial. First, the removal or value testing of the item at the head of the queue takes O(1) time. Also, although in the worst case the time to insert an item into the priority queue is of O(n), the ability to start the insertion from either end of the queue structure can help reduce the average insertion time. For example, when inserting items into the time ordered queues (nearest time at the head of the queue and farthest time at the tail) starting from the tail of the queue can result in better average insertion times. This is because time is always advancing in a real-time system and new tasks generally have starting times later than tasks currently in the system. Of course, in one particular system the opposite could be true. It is recommended that the current time ordered deques (Ready_Queue, Deadline_Queue, and Latest_Start_Time_Queue) remain as deques until some overwhelming evidence is produced to replace them with some other data structure.

6.5. Results Summary

Testing of the feasibility demonstration system has validated the approach taken. In particular, the Task Manager is able to dynamically create and control tasks as directed by the Reasoning Process. The scheduling overhead incurred is not excessive and additional algorithm and data structure improvements will reduce the scheduling overhead further. The feasibility demonstration system clearly shows the viability of the architecture developed.

However, some performance gains are possible. The task control buffer's internal data structures are inappropriate in the current implementation. The same can be said for the Tasks_By_ID_Queue. Changing these data structures to binary trees would improve the performance of these heavily used data structures. In addition, an some effort should be expended to reduce the number of context switches required in the current implementation.

VII. Conclusion

This research has taken the first steps of a much larger research effort into the development of Intelligent Real-Time Systems. This chapter outlines the specific accomplishments of this thesis effort and lays out some possible directions in which to continue the development of the architecture.

7.1. Summary

The primary result of this thesis effort is an intelligent real-time system architecture and top level design. The majority of the effort expended in this thesis has gone into researching existing work in the field and identifying what is required or common in most intelligent real-time systems and what appears to be missing or needed. The architecture developed allows for inclusion of all the identified components. Additionally, the implemented portions of the Task Manager directly address what is missing from other intelligent real-time systems, namely guaranteed ability to meet task deadlines. Adding this ability to a dynamic system involved development of the ability to dynamically create and control Ada tasks, dynamically assign priorities to those tasks, and export task scheduling and execution controls to a 'Reasoning Process', while maintaining a large degree of parallelism. The top level design and a robust Task Manager has been implemented and the architecture's feasibility demonstrated.

Overcoming the problems associated with dynamic task creation and control in Ada is fundamental to the development of an intelligent real-time system implemented in Ada. The method developed has demonstrated the ability to solve a large number of the problems involved with dynamic task creation. Namely, it allows for re-use of Ada task shells and helps prevent a potentially serious memory leak. Additionally, communication between the dynamically created tasks and the rest of the system has been addressed and a method using a "variables buffer" has been implemented and demonstrated. The methods developed to control the execution of the dynamically created tasks "appear to work well" in the limited tests conducted. However, the limited testing leaves unanswered a large number of questions regarding the efficiency of the methods developed.

Perhaps the single most important control implemented is the dynamic execution priority assignment scheme. The methods as implemented in the Task Manager are effective in solving the problems posed in guaranteeing real-time performance. The system is able to schedule and execute both periodic and non-

periodic tasks in both overloaded and non-overloaded situations, thus demonstrating an ability to degrade gracefully. To add this capability, non-standard Ada had to be used. Ada does not provide for dynamic task priority assignments and thus Verdix Ada specific procedures had to be used. The use of Verdix Ada specific procedures and functions limits portability of the system. However, this problem of dynamic task priority assignments in Ada is being addressed and may change in the near future [@Ada9X]. If and when that happens, the methods used here will have to be reevaluated and most likely changed.

Exporting task scheduling and execution controls also was demonstrated and implemented. The ability of the Reasoning Process to affect the current scheduling policies used without having to actually implement the task scheduling itself allows the Reasoning Process to operate at a conceptually higher level. Yet it can directly affect the execution of any particular task as it deems necessary. In effect, the Reasoning Process is able to perform off-line scheduling while the Task Manager performs the on-line scheduling. Again, the lack of an ability to "rigorously test" the implementation prevents optimizations to reduce the overhead required to implement such abilities or characterize their behaviors.

Finally, the developed architecture has intentionally not sacrificed any inherent parallelism in order to achieve some domain specific performance goals. The architecture developed here is easily implementable on a multi-processor system with little, if any, re-writing of the code. The cost of maintaining this parallelism is increased code complexity and the associated communication overhead.

The architecture as developed in this thesis effort is by no means the ideal one. Instead, this thesis has presented one architecture and implementation and by doing so has charted a course for future work in the field. The work done is leading edge research with a large number of unanswered questions and potential problems. Like most difficult problems, this thesis effort has provided an incremental advance towards a solution.

7.2. Recommendations

Since this research is not an end in itself, perhaps, the most important section is this one. The recommendations presented here outline the research areas that still need to be addressed to fully implement the developed architecture. Additionally, another potential path to achieve the same goal was previously mentioned in Chapter 3 but not explored by this thesis: that path also deserves to be examined. The following recommendations are divided into three parts: recommendations to improve the implemented

Task Manager, architecture components other than the Task Manager, and other implementation and development issues.

7.2.1. Task Manager Recommendations. The Task Manager as implemented can easily be improved upon. First, the system as implemented does not make use of the full rate monotonic theory. In particular, the blocking time any particular task may suffer is not included in the current periodic task utilization calculations (Theorems 3 and 4 from Chapter 2). The effect is reduced accuracy in the utilization calculations. Also, no allowance for aperiodic tasks is implemented, but details of how to incorporate support for aperiodic tasks are readily available and should be incorporated into the implementation [Sprunt, 1990].

The issues associated with determining the feasibility of the non-periodic tasks has not been adequately addressed. As implemented, priorities of non-periodic tasks are assigned earliest deadline first, and feasibility is only checked for tasks in isolation, not in conjunction with other non-periodic tasks of varying importances. Incorporation of more robust (and correspondingly more time and space complex) algorithms is required. Again, there exists a large body of knowledge dealing with these types of algorithms to draw upon and allowances for incorporating these algorithms has been made in the existing design [Liu, 1991] [Coffman, 1976].

Also, although memory management has been addressed, the problem has by no means been solved. In particular, each data structure retains the maximum memory it has ever been allotted. This approach may lead to problems when the system operates in an overloaded condition. There is currently no method implemented to deal with Ada storage errors generated when no more memory is available. It is possible for this condition to occur and a critical task is unable to execute. A method must be implemented to allow for the reclamation of memory when required.

Finally, the problems with using the Ada delay statement for accurate timing control are well known in the Ada community. The Task Dispatcher, as implemented, makes use of the Ada delay statement and should be corrected. Attaching a procedure to directly respond to timer generated interrupts should greatly enhance the timing accuracy of the Task Dispatcher.

7.2.2. Other Architecture Components Recommendations. Neither the Environment Model, System Model, or the reasoning logic of the Reasoning Process was implemented in this thesis effort, primarily because they all appear to be very domain specific. Choosing a domain and developing the reasoning logic,

Environment Model, and System Model for that domain should be attempted. Choosing a particular domain should greatly simplify the development effort by anchoring the system to some definable performance criteria to measure the effectiveness of the system as a whole. Thus, not only can speed of execution be evaluated against some requirement, but also the quality of the system's responses can be evaluated.

The Environment Model should allow for efficient access by all currently executing tasks and address the issues of data consistency and data timeliness. Given time, I had envisioned using an object oriented Ada pre-processor (Classic Ada) for implementing the Environment Model. I believe an object oriented approach most suitable for this type of model but clearly the issue requires investigation. Additionally, the tradeoffs associated with placement of the data pertaining to the condition of the environment into a single repository versus distribution among the various tasks should be investigated.

The System Model should allow the Reasoning Process the ability to make effective predictions of future events and accurately reflect the current state of the system. A fundamental assumption of this thesis effort was the existence of such a model in the form of a task network or plan-goal graph. Converting these knowledge acquisition tools into an implementation usable by the reasoning process is required. The envisioned method uses a graph structure with each node representing a task in the system. Future efforts along these lines should closely examine the idea of temporal constraint networks when developing the system model and the corresponding reasoning logic [Dechter, 1991].

The current Reasoning Process makes only minimum use of existing techniques for agenda management. The architecture, however, allows for relatively easy inclusion of most of the techniques developed under the Pilot's Associate program and presented in section 2.3.3 of the thesis into design. By moving the Reasoning Process to a separate processor and including the additional agenda management techniques, I believe a significant overall performance gain can be achieved.

Finally, specific implementations of the I/O Process should be investigated to match performance with system capabilities. In particular, the idea of reflexive behavior can be implemented by allowing the I/O Process to instantiate tasks in response to external events directly, without having to report the event first to the Reasoning Process. The correct mix of this type of reactive behavior versus reasoned behavior should be investigated.

7.2.3. Implementation and Development Recommendations. The development of this thesis suffered significantly from the lack of a dedicated or single user workstation. Because the system was developed on a multi-user system, accurate timing information is virtually impossible to obtain. The multi-user system used allows other processes to start and stop at anytime. Additionally, there is the additional overhead associated with managing multiple user processes. It is highly recommended that any future effort use a dedicated workstation operating in the single user mode to provide an accurate picture of the system's performance and timing characteristics.

Although allowed for in the design, the system does not make use of multi-processors. Splitting the system across multiple processors would greatly increase the performance of the system. I would recommend moving the Reasoning Process to a separate machine as the first step. This move would alleviate the problem of which priority to assign the Reasoning Process. Additionally, it should allow the remaining components to act more as a traditional real-time system with the corresponding improvement in its ability to meet task deadlines, while at the same time, allowing the Reasoning Process more computational resources and hopefully a corresponding increase in overall system performance.

Tools for the development of these types of systems should also be explored. In particular, tools to debug multiple task Ada programs should be examined. Also useful, would be the development of a system that allows for a 'rubber clock' so that system's view of the passage of time could be slowed. The rubber clock would allow a programmer the ability to step through the program with the clock only advancing by the time to execute each step and stopping while the programmer examines the results of that step.

The final recommendation is for a more rigorous specification process to be conducted. Unfortunately, only modest software engineering techniques were applied in the development of the feasibility demonstration system. Using a structured analysis and design method would provide better measurement criteria, greater design visibility, and perhaps increased performance. Using this thesis and the accompanying code as a guide, the task could be easily accomplished as another thesis effort.

7.3. Thesis Summary

The literature review reported in Chapter 2 identified the Environment Model, System Model, I/O Process, Reasoning Process, and Task Manager as the components necessary for an intelligent real-time system. It also identified real-time task scheduling and deadline guarantees as the missing component in

most previous IRTSs. In addition, rate monotonic theory and imprecise computation scheduling were discussed as ways to handle real-time task scheduling. Finally, Chapter 2 identified the general task types any-time, singular, and periodic.

Chapters 3 discussed IRTS performance measures and design considerations. Speed, responsiveness, timeliness, graceful degradation, data consistency, and solution quality are the performance measures for intelligent real-time systems. Control reasoning, focus of attention, parallelism, and improving algorithm efficacy are design considerations used when addressing the performance measures. Chapter 3 concluded with the rationale for the design approach used.

Chapter 4 presented the top level view of an intelligent system architecture and discussed general issues of the architecture. The methods used to perform on-line task scheduling of both non-periodic and periodic tasks was presented. Chapter 5 discussed the details of the demonstration system implemented to confirm the feasibility of the architecture presented in Chapter 4 and Chapter 6 discussed the results of the feasibility demonstration.

This thesis effort has researched and developed a feasible architecture for use in creating an intelligent real-time system. Dynamic task creation and real-time scheduling methods were developed and successfully demonstrated. But the work is not complete. It is clear that further research is required along a number of different paths. This research establishes a starting point and provides possible paths for further work.

Appendix A. Test Results

This appendix contains numerous graphs that show the results of timing tests conducted on the feasibility demonstration system. The effect of the underlying UNIX multiproccssing/time slicing operating system can be clearly seen in all the graphs. The graphs are not intended to be accurate measures of the systems performance, rather they are intended to validate the approach taken in this research. In addition, a number of "queue dumps" are included that show the effects of adding, modifying, or removing tasks from the Task Manager.

A.1 Scheduling Overhead Timing Results

Figures A.1 through A.4 show a sample of the measured times to add, modify, and remove periodic tasks. For these tests, a loop which first added a new task, then modified it was executed 150 tasks. Once all 150 tasks were created, another loop was executed which removed all 150. Both loops were run 20 times and the minimum, maximum, and average times for each operation recorded. The vertical lines represent the range of times obtained with the corresponding number of acitve tasks in the system. The tick marks on each line represent the minimum, average, and maximum times. For these tests, there were no non-periodic tasks currently active in the system.

Figures A.5 through A.8 show a sample of the measured times to add, modify, and remove non-periodic tasks. The same testing method used for the periodic task tests was used to acquire these times. For these tests, there were no periodic tasks currently active in the system.

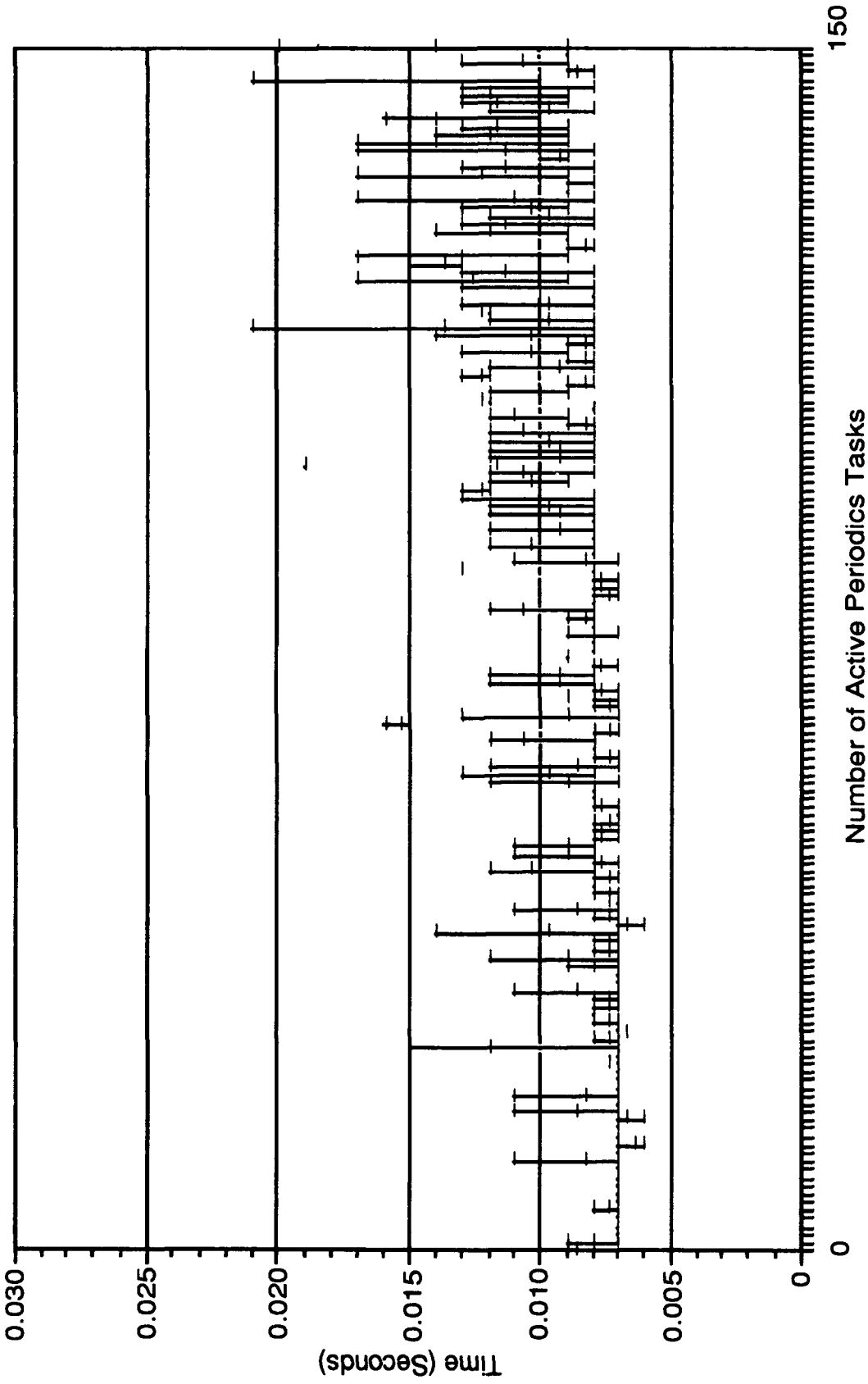

Figure A.1 Periodic Tasks Add Time, New Task Instantiated

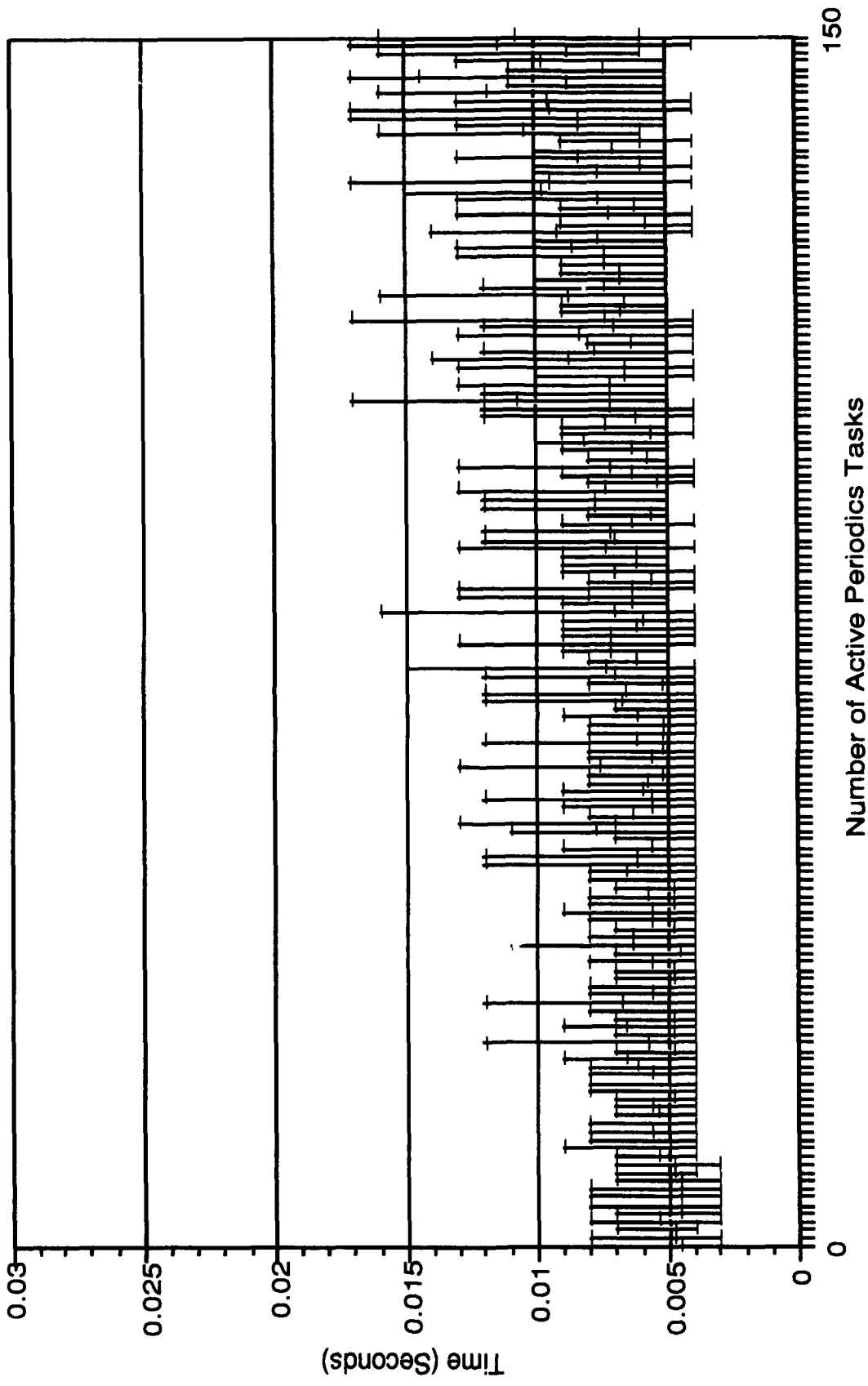

Figure A.2 Periodic Tasks Add Time, Task Shell Reused

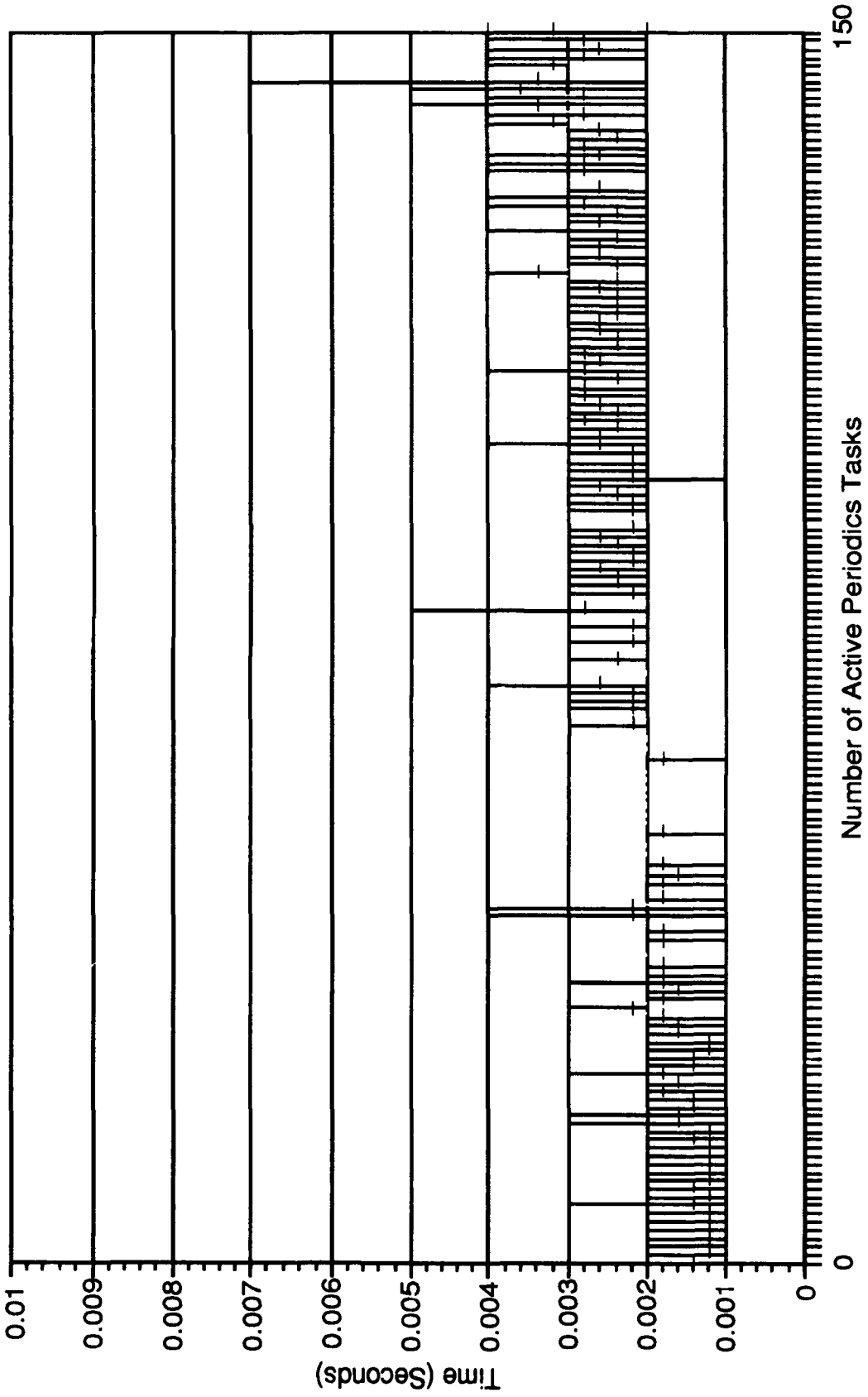

Figure A.3 Periodic Tasks Modify Time, Period and Importance Changed

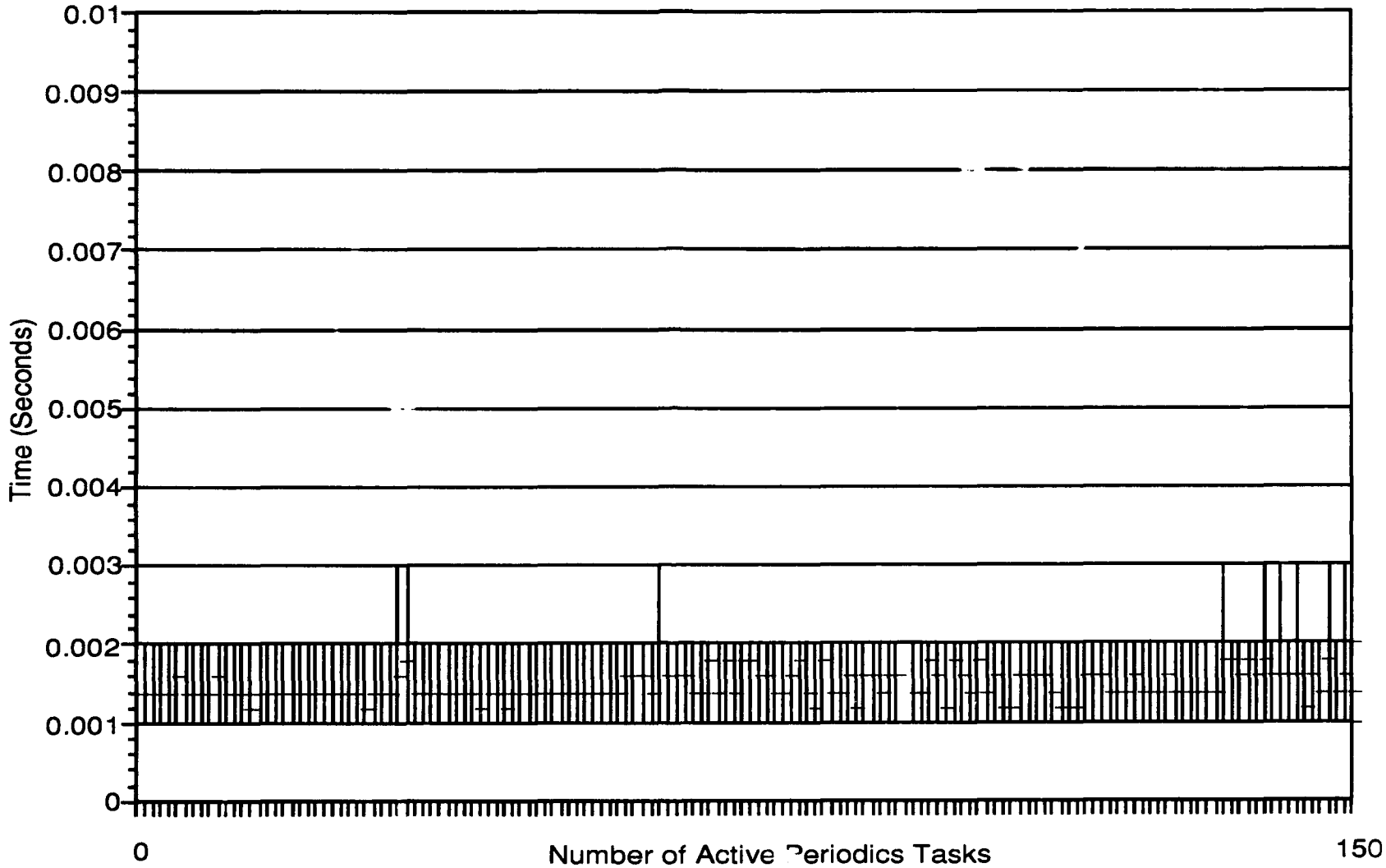

Figure A.4 Periodic Tasks Remove Time

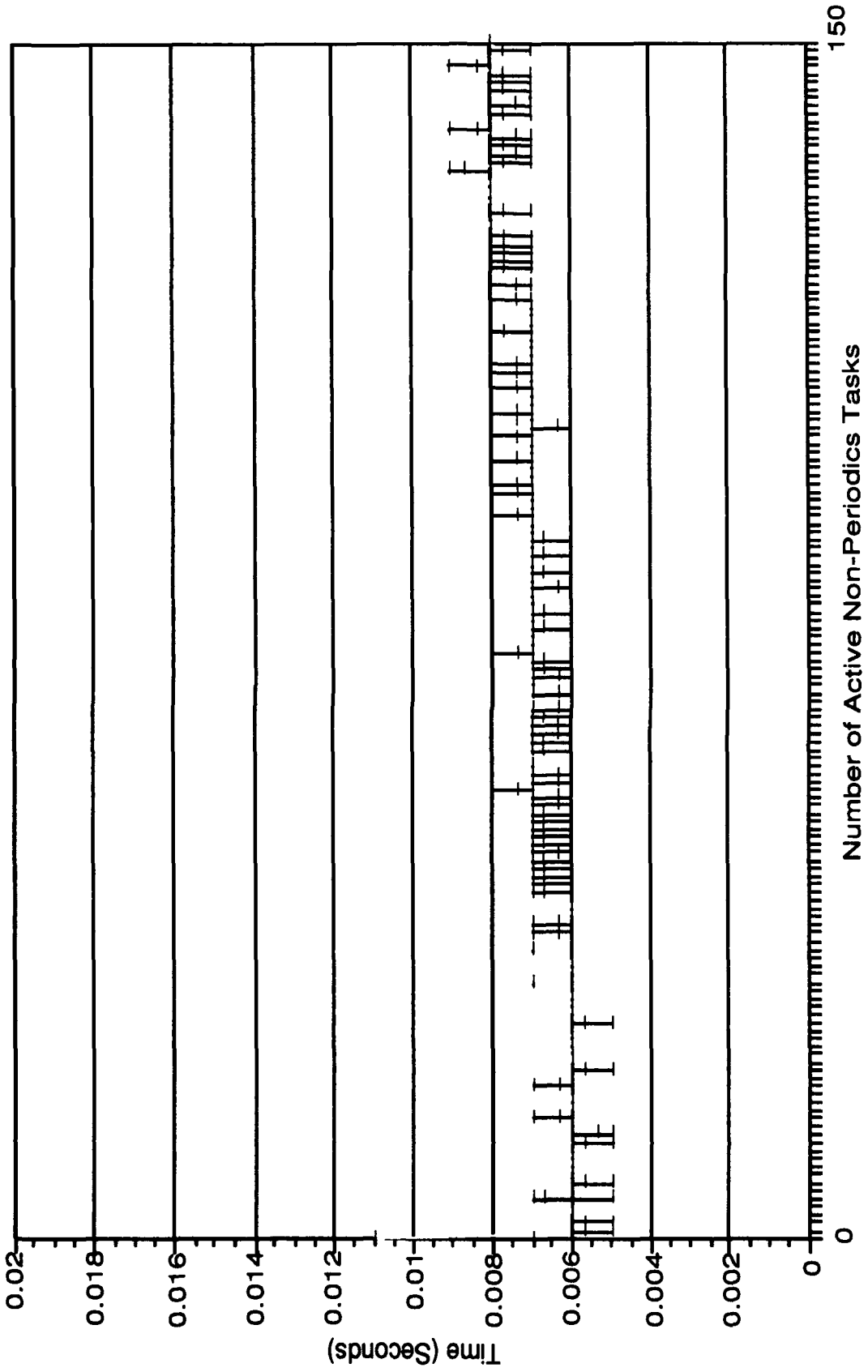

Figure A.5 Non-Periodic Tasks Add Time, New Task Instantiated

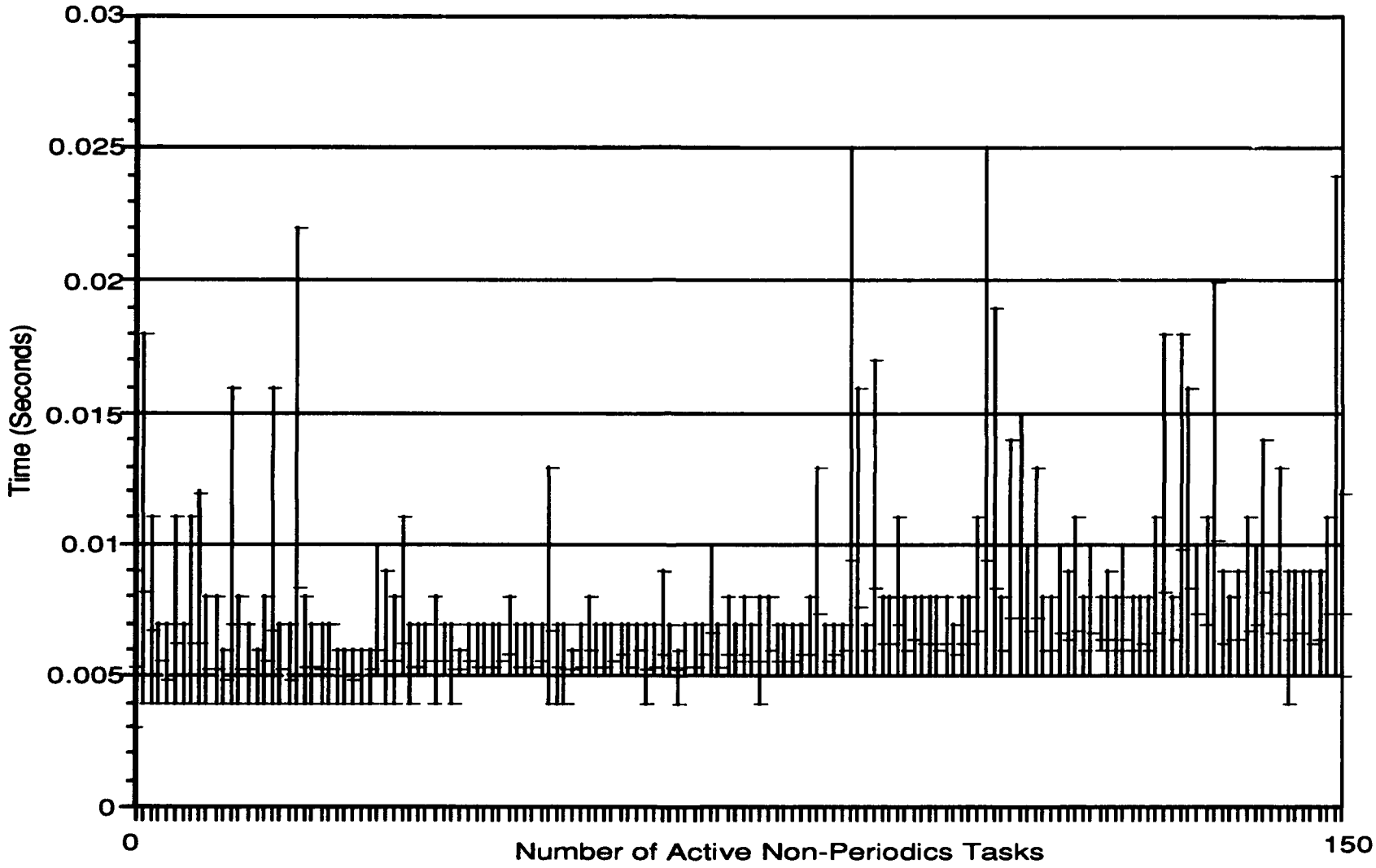

Figure A.6 Non-Periodic Tasks Add Time, Task Shell Reused

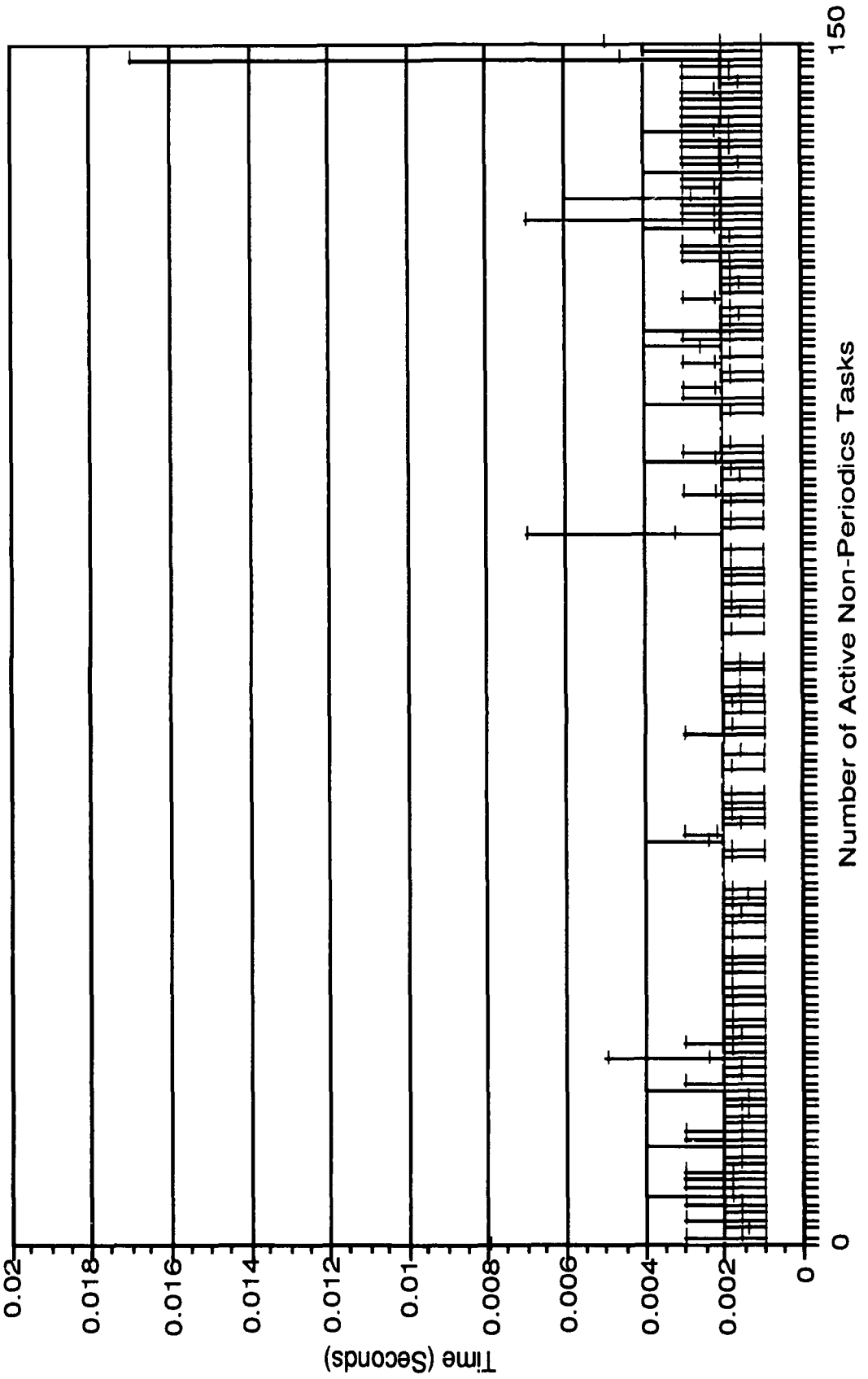

Figure A.7 Non-Periodic Tasks Modify Time, Deadline and Importance Changed

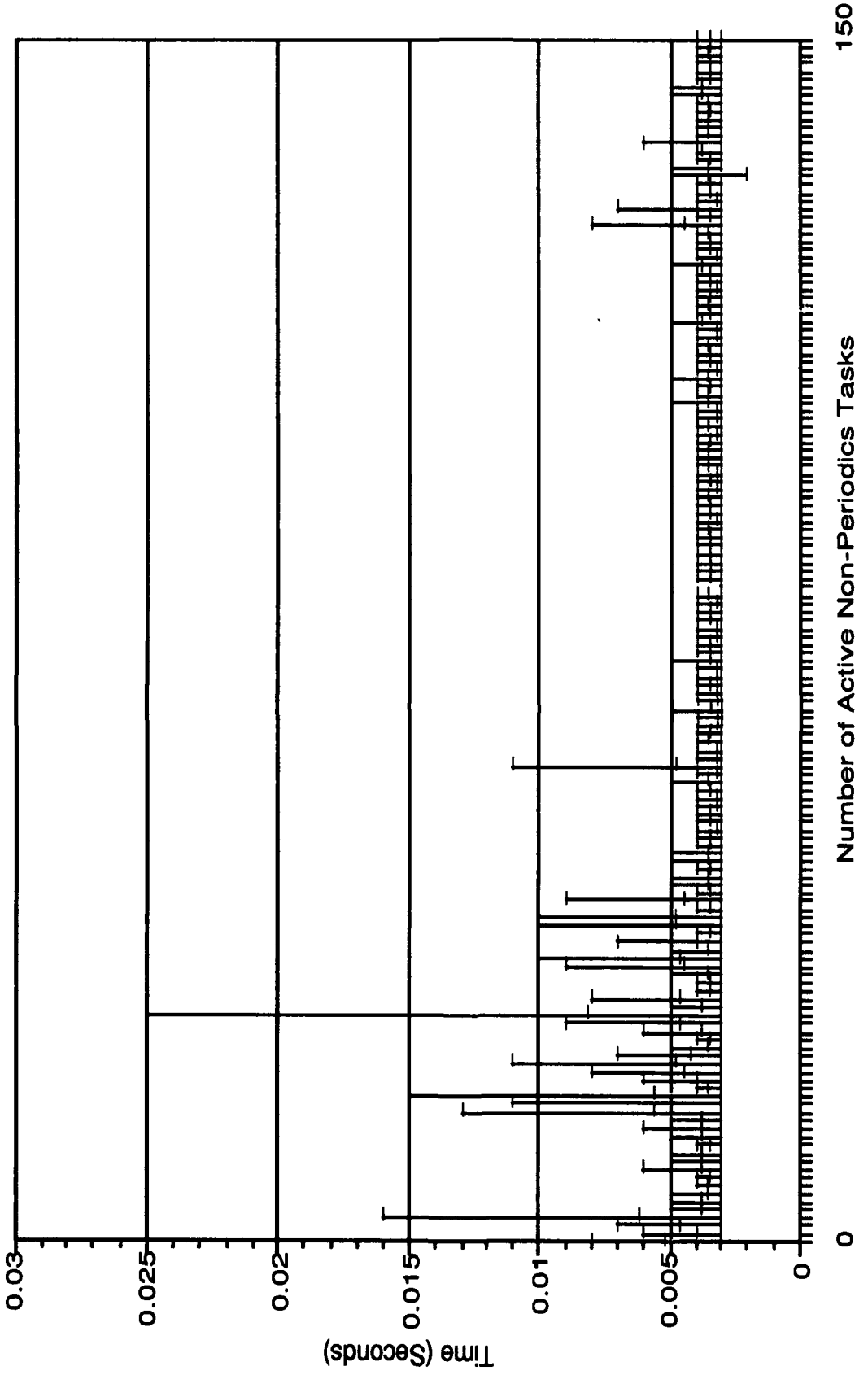

Figure A.8 Non-Periodic Tasks Remove Time

A.2. Schedules Produced

The following sample printouts show the affect of adding a periodic task whose additional utilization causes the the periodic condition to change from ALL_OPTIONAL to SOME_OPTIONAL. Periodic task kind 8 has mandatory duration of 0.008 seconds and an optional duration of 0.02 seconds. The first printout shows the state of the periodic task set before the addition of Task ID 1280600 and the second one shows the state of the periodic task set after Task ID 1280600 has been added. Note some tasks have been deleted from the printout to make it easier to read.

Task ID	Period	Kind	Importance	Priority	Mode
1233880	1.12100	8	46	90	OPTIONAL
848440	1.49900	8	13	89	OPTIONAL
976920	2.29800	8	24	88	OPTIONAL
1245560	2.39900	8	47	87	OPTIONAL
790040	2.43300	8	8	86	OPTIONAL
965240	2.59300	8	23	85	OPTIONAL
801720	3.42200	8	9	84	OPTIONAL
930200	3.78900	8	20	83	OPTIONAL
731640	3.93600	8	3	82	OPTIONAL
1058680	4.23600	8	31	81	OPTIONAL
1257240	4.34100	8	48	80	OPTIONAL
836760	4.36300	8	12	79	OPTIONAL
1035320	4.45700	8	29	78	OPTIONAL
860120	4.49800	8	14	77	OPTIONAL
1011960	4.60500	8	27	76	OPTIONAL
918520	4.96800	8	19	75	OPTIONAL
719960	5.13200	8	2	74	OPTIONAL
825080	5.40600	8	11	73	OPTIONAL
906840	5.57900	8	18	72	OPTIONAL
883480	5.64100	8	16	71	OPTIONAL
941880	5.90700	8	21	70	OPTIONAL
988600	6.09400	8	25	69	OPTIONAL
871800	6.61600	8	15	68	OPTIONAL
1187160	7.15300	8	42	67	OPTIONAL
1082040	7.47600	8	33	66	OPTIONAL
1047000	7.48300	8	30	65	OPTIONAL
1128760	7.51200	8	37	64	OPTIONAL
895160	7.52000	8	17	63	OPTIONAL
1210520	7.64000	8	44	62	OPTIONAL
1222200	7.68000	8	45	61	OPTIONAL
1000280	7.92000	8	26	60	OPTIONAL
1093720	8.14400	8	34	59	OPTIONAL
1070360	8.21900	8	32	58	OPTIONAL
1140440	8.24400	8	38	57	OPTIONAL
1152120	8.38500	8	39	56	OPTIONAL
778360	8.86800	8	7	55	OPTIONAL
1117080	8.90200	8	36	54	OPTIONAL
953560	8.92700	8	22	53	OPTIONAL
755000	8.97900	8	5	52	OPTIONAL
743320	9.37000	8	4	51	OPTIONAL
1175480	9.79600	8	41	50	OPTIONAL
813400	9.82100	8	10	49	OPTIONAL
1163800	10.03100	8	40	48	OPTIONAL
1268920	10.12300	8	49	47	OPTIONAL

1105400	10.13400	8	35	46	OPTIONAL
1198840	10.46100	8	43	45	OPTIONAL
766680	10.68200	8	6	44	OPTIONAL
1023640	10.76600	8	28	43	OPTIONAL

Periodic Utilization => 2.75986964545906E-01
Mandatory Utilization => 8.23119017066736E-02
Optional Utilization => 2.75986964545906E-01
Required Utilization => 2.79270430800456E-01
Periodic Utilization Budget => 4.00000000000000E-01
Current Periodic Condition => ALL_OPTIONAL

Task ID	Period	Kind	Importance	Priority	Mode
1233880	1.12100	8	46	90	OPTIONAL
848440	1.49900	8	13	89	OPTIONAL
976920	2.29800	8	24	88	OPTIONAL
1245560	2.39900	8	47	87	OPTIONAL
790040	2.43300	8	8	86	OPTIONAL
1280600	2.55800	8	49	85	MANDATORY
965240	2.59300	8	23	84	OPTIONAL
801720	3.42200	8	9	83	OPTIONAL
930200	3.78900	8	20	82	OPTIONAL
731640	3.93600	8	3	81	OPTIONAL
1058680	4.23600	8	31	80	OPTIONAL
1257240	4.34100	8	48	79	MANDATORY
836760	4.36300	8	12	78	OPTIONAL
1035320	4.45700	8	29	77	OPTIONAL
860120	4.49800	8	14	76	OPTIONAL
1011960	4.60500	8	27	75	OPTIONAL
918520	4.96800	8	19	74	OPTIONAL
719960	5.13200	8	2	73	OPTIONAL
825080	5.40600	8	11	72	OPTIONAL
906840	5.57900	8	18	71	OPTIONAL
883480	5.64100	8	16	70	OPTIONAL
941880	5.90700	8	21	69	OPTIONAL
988600	6.09400	8	25	68	OPTIONAL
871800	6.61600	8	15	67	OPTIONAL
1187160	7.15300	8	42	66	OPTIONAL
1082040	7.47600	8	33	65	OPTIONAL
1047000	7.48300	8	30	64	OPTIONAL
1128760	7.51200	8	37	63	OPTIONAL
895160	7.52000	8	17	62	OPTIONAL
1210520	7.64000	8	44	61	OPTIONAL
1222200	7.68000	8	45	60	OPTIONAL
1000280	7.92000	8	26	59	OPTIONAL
1093720	8.14400	8	34	58	OPTIONAL
1070360	8.21900	8	32	57	OPTIONAL
1140440	8.24400	8	38	56	OPTIONAL
1152120	8.38500	8	39	55	OPTIONAL
778360	8.86800	8	7	54	OPTIONAL
1117080	8.90200	8	36	53	OPTIONAL
953560	8.92700	8	22	52	OPTIONAL
755000	8.97900	8	5	51	OPTIONAL
743320	9.37000	8	4	50	OPTIONAL
1175480	9.79600	8	41	49	OPTIONAL
813400	9.82100	8	10	48	OPTIONAL
1163800	10.03100	8	40	47	OPTIONAL
1268920	10.12300	8	49	46	MANDATORY
1105400	10.13400	8	35	45	OPTIONAL
1198840	10.46100	8	43	44	OPTIONAL
766680	10.68200	8	6	43	OPTIONAL
1023640	10.76600	8	28	42	OPTIONAL

```
Periodic Utilization  =>  2.77404244072816E-01
Mandatory Utilization =>  8.56348102289566E-02
Optional Utilization  =>  2.87128481355913E-01
Required Utilization  =>  2.79229184559615E-01
Periodic Utilization Budget =>  4.00000000000000E-01
Current Periodic Condition  =>  SOME_OPTIONAL
```

The following printouts show the affects of modifying a periodic task. In this case, the period of Task ID 801720 has been changed from 2.422 seconds to 3.422 seconds and its importance value decreased by one. Note its priority has been changed to reflect its new period along with the priority of Task ID 790040.

Task ID	Period	Kind	Importance	Priority	Mode
801720	2.42200	8	8	90	OPTIONAL
790040	2.43300	8	8	89	OPTIONAL
731640	3.93600	8	3	88	OPTIONAL
719960	5.13200	8	2	87	OPTIONAL
778360	8.86800	8	7	86	OPTIONAL
755000	8.97900	8	5	85	OPTIONAL
743320	9.37000	8	4	84	OPTIONAL
766680	10.68200	8	6	83	OPTIONAL

```
Periodic Utilization  =>  4.83728493284041E-02
Mandatory Utilization =>  1.44269901505767E-02
Optional Utilization  =>  4.83728493284041E-02
Required Utilization  =>  2.89624744528825E-01
Periodic Utilization Budget =>  4.00000000000000E-01
Current Periodic Condition  =>  ALL_OPTIONAL
```

Task ID	Period	Kind	Importance	Priority	Mode
790040	2.43300	8	8	90	OPTIONAL
801720	3.42200	8	9	89	OPTIONAL
731640	3.93600	8	3	88	OPTIONAL
719960	5.13200	8	2	87	OPTIONAL
778360	8.86800	8	7	86	OPTIONAL
755000	8.97900	8	5	85	OPTIONAL
743320	9.37000	8	4	84	OPTIONAL
766680	10.68200	8	6	83	OPTIONAL

```
Periodic Utilization  =>  4.49341776161001E-02
Mandatory Utilization =>  1.34014213942755E-02
Optional Utilization  =>  4.49341776161001E-02
Required Utilization  =>  2.89624744528825E-01
Periodic Utilization Budget =>  4.00000000000000E-01
Current Periodic Condition  =>  ALL_OPTIONAL
```

The following printout traces modify operations on an active set of any-time tasks. There are two sets of data for each modify operation, one taken immediately before the modify, and one immediately after. During the modify, a task's deadline and importance is changed. The numbers in the status column represent the task's current state. The number to state translation is as shown in Table A.1. Note that at times, the time remaining for a task jumps back up to 0.100 seconds. The reason for the jumps is the

method used to handle errors induced by the UNIX multitasking operating system. In order to insure that the time remaining never goes negative, a check in the code detects this situation and sets the time remaining to 0.100 seconds.

Table A.1 Printout Status Number to State Name Translation

Status Number	State Name	Applies to
0	READY	All
1	EXECUTING	periodic tasks
2	EXECUTING_MANDATORY	non-periodic tasks
3	EXECUTING_OPTIONAL	non-periodic tasks
4	PREEMPTED_MANDATORY	non-periodic tasks
5	PREEMPTED_OPTIONAL	non-periodic tasks
6	DISCARDED	non-periodic tasks
7	COMPLETED	non-periodic tasks

```
  >>>>>>>>>  The Time Now is =>     60175.427  Periodic Utilization Budget => 5.00E-01 <<<<<<<<<
               Durations          Time                                                    Latest
 Task ID   Mandatory Optional Remaining Start-Time  Deadline  Started-At Importance Status Start-Time
---------- --------- -------- --------- ---------- ---------- ---------- ---------- ------ ----------
   749552     0.100    1.000     0.200   60175.363  60178.886  60175.422          1      2


  >>>>>>>>>  The Time Now is =>     60175.453  Periodic Utilization Budget => 5.00E-01 <<<<<<<<<
               Durations          Time                                                    Latest
 Task ID   Mandatory Optional Remaining Start-Time  Deadline  Started-At Importance Status Start-Time
---------- --------- -------- --------- ---------- ---------- ---------- ---------- ------ ----------
   749552     0.100    1.000     0.178   60175.363  60179.886  60175.422          2      2


  >>>>>>>>>  The Time Now is =>     60175.727  Periodic Utilization Budget => 5.00E-01 <<<<<<<<<
               Durations          Time                                                    Latest
 Task ID   Mandatory Optional Remaining Start-Time  Deadline  Started-At Importance Status Start-Time
---------- --------- -------- --------- ---------- ---------- ---------- ---------- ------ ----------
   749552     0.100    1.000     1.760   60175.363  60179.886     N/A             2      5  60178.126
   761232     0.100    1.000     0.200   60175.674  60185.718  60175.725          2      2


  >>>>>>>>>  The Time Now is =>     60175.757  Periodic Utilization Budget => 5.00E-01 <<<<<<<<<
               Durations          Time                                                    Latest
 Task ID   Mandatory Optional Remaining Start-Time  Deadline  Started-At Importance Status Start-Time
---------- --------- -------- --------- ---------- ---------- ---------- ---------- ------ ----------
   749552     0.100    1.000     1.760   60175.363  60179.886     N/A             2      5  60178.126
   761232     0.100    1.000     0.178   60175.674  60186.718  60175.725          3      2


  >>>>>>>>>  The Time Now is =>     60176.139  Periodic Utilization Budget => 5.00E-01 <<<<<<<<<
               Durations          Time                                                    Latest
 Task ID   Mandatory Optional Remaining Start-Time  Deadline  Started-At Importance Status Start-Time
---------- --------- -------- --------- ---------- ---------- ---------- ---------- ------ ----------
   749552     0.100    1.000     1.321   60175.363  60179.886     N/A             2      5  60178.565
   761232     0.100    1.000     2.000   60175.674  60186.718     N/A             3      5  60184.718
   772912     0.100    1.000     0.200   60175.983  60185.558  60176.135          3      2
```

>>>>>>>>> The Time Now is => 60176.169 Periodic Utilization Budget => 5.00E-01 <<<<<<<<<

Task ID	Durations Mandatory	Optional	Time Remaining	Start-Time	Deadline	Started-At	Importance	Status	Latest Start-Time
749552	0.100	1.000	1.321	60175.363	60179.886	N/A	2	5	60178.565
761232	0.100	1.000	2.000	60175.674	60186.718	N/A	3	5	60184.718
772912	0.100	1.000	0.178	60175.983	60186.558	60176.135	4	2	

>>>>>>>>> The Time Now is => 60176.431 Periodic Utilization Budget => 5.00E-01 <<<<<<<<<

Task ID	Durations Mandatory	Optional	Time Remaining	Start-Time	Deadline	Started-At	Importance	Status	Latest Start-Time
749552	0.100	1.000	1.122	60175.363	60179.886	N/A	2	5	60178.764
761232	0.100	1.000	2.000	60175.674	60186.718	N/A	3	5	60184.718
772912	0.100	1.000	2.000	60175.983	60186.558	N/A	4	5	60184.558
784592	0.100	1.000	0.200	60176.392	60188.011	60176.426	4	2	

>>>>>>>>> The Time Now is => 60176.474 Periodic Utilization Budget => 5.00E-01 <<<<<<<<<

Task ID	Durations Mandatory	Optional	Time Remaining	Start-Time	Deadline	Started-At	Importance	Status	Latest Start-Time
749552	0.100	1.000	1.122	60175.363	60179.886	N/A	2	5	60178.764
761232	0.100	1.000	2.000	60175.674	60186.718	N/A	3	5	60184.718
772912	0.100	1.000	2.000	60175.983	60186.558	N/A	4	5	60184.558
784592	0.100	1.000	0.163	60176.392	60189.011	60176.426	5	2	

>>>>>>>>> The Time Now is => 60176.735 Periodic Utilization Budget => 5.00E-01 <<<<<<<<<

Task ID	Durations Mandatory	Optional	Time Remaining	Start-Time	Deadline	Started-At	Importance	Status	Latest Start-Time
749552	0.100	1.000	0.909	60175.363	60179.886	N/A	2	5	60178.977
761232	0.100	1.000	2.000	60175.674	60186.718	N/A	3	5	60184.718
772912	0.100	1.000	2.000	60175.983	60186.558	N/A	4	5	60184.558
784592	0.100	1.000	2.000	60176.392	60189.011	N/A	5	5	60187.011
796272	0.100	1.000	0.200	60176.713	60186.154	60176.734	5	2	

>>>>>>>>> The Time Now is => 60176.749 Periodic Utilization Budget => 5.00E-01 <<<<<<<<<

Task ID	Durations Mandatory	Optional	Time Remaining	Start-Time	Deadline	Started-At	Importance	Status	Latest Start-Time
749552	0.100	1.000	0.909	60175.363	60179.886	N/A	2	5	60178.977
761232	0.100	1.000	2.000	60175.674	60186.718	N/A	3	5	60184.718
772912	0.100	1.000	2.000	60175.983	60186.558	N/A	4	5	60184.558
784592	0.100	1.000	2.000	60176.392	60189.011	N/A	5	5	60187.011
796272	0.100	1.000	0.187	60176.713	60187.154	60176.734	6	2	

>>>>>>>>> The Time Now is => 60177.109 Periodic Utilization Budget => 5.00E-01 <<<<<<<<<

Task ID	Durations Mandatory	Optional	Time Remaining	Start-Time	Deadline	Started-At	Importance	Status	Latest Start-Time
749552	0.100	1.000	0.749	60175.363	60179.886	N/A	2	5	60179.137
761232	0.100	1.000	2.000	60175.674	60186.718	N/A	3	5	60184.718
772912	0.100	1.000	2.000	60175.983	60186.558	N/A	4	5	60184.558
784592	0.100	1.000	2.000	60176.392	60189.011	N/A	5	5	60187.011
796272	0.100	1.000	2.000	60176.713	60187.154	N/A	6	5	60185.154
807952	0.100	1.000	0.200	60177.015	60178.735	60177.108	6	2	

>>>>>>>>> The Time Now is => 60177.153 Periodic Utilization Budget => 5.00E-01 <<<<<<<<<

Task ID	Durations Mandatory	Optional	Time Remaining	Start-Time	Deadline	Started-At	Importance	Status	Latest Start-Time
749552	0.100	1.000	0.749	60175.363	60179.886	N/A	2	5	60179.137
761232	0.100	1.000	2.000	60175.674	60186.718	N/A	3	5	60184.718
772912	0.100	1.000	2.000	60175.983	60186.558	N/A	4	5	60184.558
784592	0.100	1.000	2.000	60176.392	60189.011	N/A	5	5	60187.011
796272	0.100	1.000	2.000	60176.713	60187.154	N/A	6	5	60185.154
807952	0.100	1.000	0.165	60177.015	60179.735	60177.108	7	2	

>>>>>>>>> The Time Now is => 60177.415 Periodic Utilization Budget => 5.00E-01 <<<<<<<<<

Task ID	Durations Mandatory	Optional	Time Remaining	Start-Time	Deadline	Started-At	Importance	Status	Latest Start-Time
749552	0.100	1.000	0.749	60175.363	60179.886	N/A	2	5	60179.137
761232	0.100	1.000	2.000	60175.674	60186.718	N/A	3	5	60184.718
772912	0.100	1.000	2.000	60175.983	60186.558	N/A	4	5	60184.558
784592	0.100	1.000	2.000	60176.392	60189.011	N/A	5	5	60187.011
796272	0.100	1.000	2.000	60176.713	60187.154	N/A	6	5	60185.154
807952	0.100	1.000	1.777	60177.015	60179.735	N/A	7	5	60177.958
819632	0.100	1.000	0.200	60177.374	60180.281	60177.414	7	2	

>>>>>>>>> The Time Now is => 60177.432 Periodic Utilization Budget => 5.00E-01 <<<<<<<<<

Task ID	Durations Mandatory	Optional	Time Remaining	Start-Time	Deadline	Started-At	Importance	Status	Latest Start-Time
749552	0.100	1.000	0.749	60175.363	60179.886	N/A	2	5	60179.137
761232	0.100	1.000	2.000	60175.674	60186.718	N/A	3	5	60184.718
772912	0.100	1.000	2.000	60175.983	60186.558	N/A	4	5	60184.558
784592	0.100	1.000	2.000	60176.392	60189.011	N/A	5	5	60187.011
796272	0.100	1.000	2.000	60176.713	60187.154	N/A	6	5	60185.154
807952	0.100	1.000	1.777	60177.015	60179.735	N/A	7	5	60177.958
819632	0.100	1.000	0.185	60177.374	60181.281	60177.414	8	2	

>>>>>>>>> The Time Now is => 60177.729 Periodic Utilization Budget => 5.00E-01 <<<<<<<<<

Task ID	Durations Mandatory	Optional	Time Remaining	Start-Time	Deadline	Started-At	Importance	Status	Latest Start-Time
749552	0.100	1.000	0.749	60175.363	60179.886	N/A	2	5	60179.137
761232	0.100	1.000	2.000	60175.674	60186.718	N/A	3	5	60184.718
772912	0.100	1.000	2.000	60175.983	60186.558	N/A	4	5	60184.558
784592	0.100	1.000	2.000	60176.392	60189.011	N/A	5	5	60187.011
796272	0.100	1.000	2.000	60176.713	60187.154	N/A	6	5	60185.154
807952	0.100	1.000	1.417	60177.015	60179.735	N/A	7	5	60178.318
819632	0.100	1.000	2.000	60177.374	60181.281	N/A	8	5	60179.281
831312	0.100	1.000	0.200	60177.651	60188.236	60177.728	8	2	

>>>>>>>>> The Time Now is => 60177.748 Periodic Utilization Budget => 5.00E-01 <<<<<<<<<

Task ID	Durations Mandatory	Optional	Time Remaining	Start-Time	Deadline	Started-At	Importance	Status	Latest Start-Time
749552	0.100	1.000	0.749	60175.363	60179.886	N/A	2	5	60179.137
761232	0.100	1.000	2.000	60175.674	60186.718	N/A	3	5	60184.718
772912	0.100	1.000	2.000	60175.983	60186.558	N/A	4	5	60184.558
784592	0.100	1.000	2.000	60176.392	60189.011	N/A	5	5	60187.011
796272	0.100	1.000	2.000	60176.713	60187.154	N/A	6	5	60185.154
807952	0.100	1.000	1.417	60177.015	60179.735	N/A	7	5	60178.318
819632	0.100	1.000	2.000	60177.374	60181.281	N/A	8	5	60179.281
831312	0.100	1.000	0.183	60177.651	60189.236	60177.728	9	2	

>>>>>>>>> The Time Now is => 60178.025 Periodic Utilization Budget => 5.00E-01 <<<<<<<<<

Task ID	Durations Mandatory	Optional	Time Remaining	Start-Time	Deadline	Started-At	Importance	Status	Latest Start-Time
749552	0.100	1.000	0.749	60175.363	60179.886	N/A	2	5	60179.137
761232	0.100	1.000	2.000	60175.674	60186.718	N/A	3	5	60184.718
772912	0.100	1.000	2.000	60175.983	60186.558	N/A	4	5	60184.558
784592	0.100	1.000	2.000	60176.392	60189.011	N/A	5	5	60187.011
796272	0.100	1.000	2.000	60176.713	60187.154	N/A	6	5	60185.154
807952	0.100	1.000	1.111	60177.015	60179.735	N/A	7	5	60178.624
819632	0.100	1.000	2.000	60177.374	60181.281	N/A	8	5	60179.281
831312	0.100	1.000	2.000	60177.651	60189.236	N/A	9	5	60187.236
842992	0.100	1.000	0.200	60177.971	60183.258	60178.022	9	2	

>>>>>>>>> The Time Now is => 60178.069 Periodic Utilization Budget => 5.00E-01 <<<<<<<<<

Task ID	Durations Mandatory	Optional	Time Remaining	Start-Time	Deadline	Started-At	Importance	Status	Latest Start-Time
749552	0.100	1.000	0.749	60175.363	60179.886	N/A	2	5	60179.137
761232	0.100	1.000	2.000	60175.674	60186.718	N/A	3	5	60184.718
772912	0.100	1.000	2.000	60175.983	60186.558	N/A	4	5	60184.558
784592	0.100	1.000	2.000	60176.392	60189.011	N/A	5	5	60187.011
796272	0.100	1.000	2.000	60176.713	60187.154	N/A	6	5	60185.154
807952	0.100	1.000	1.111	60177.015	60179.735	N/A	7	5	60178.624
819632	0.100	1.000	2.000	60177.374	60181.281	N/A	8	5	60179.281
831312	0.100	1.000	2.000	60177.651	60189.236	N/A	9	5	60187.236
842992	0.100	1.000	0.164	60177.971	60184.258	60178.022	10	2	

>>>>>>>>> The Time Now is => 60178.335 Periodic Utilization Budget => 5.00E-01 <<<<<<<<<

Task ID	Durations Mandatory	Optional	Time Remaining	Start-Time	Deadline	Started-At	Importance	Status	Latest Start-Time
749552	0.100	1.000	0.749	60175.363	60179.886	N/A	2	5	60179.137
761232	0.100	1.000	2.000	60175.674	60186.718	N/A	3	5	60184.718
772912	0.100	1.000	2.000	60175.983	60186.558	N/A	4	5	60184.558
784592	0.100	1.000	2.000	60176.392	60189.011	N/A	5	5	60187.011
796272	0.100	1.000	2.000	60176.713	60187.154	N/A	6	5	60185.154
807952	0.100	1.000	0.926	60177.015	60179.735	N/A	7	5	60178.809
819632	0.100	1.000	2.000	60177.374	60181.281	N/A	8	5	60179.281
831312	0.100	1.000	2.000	60177.651	60189.236	N/A	9	5	60187.236
842992	0.100	1.000	2.000	60177.971	60184.258	N/A	10	5	60182.258
854672	0.100	1.000	0.200	60178.301	60182.336	60178.324	10	2	

>>>>>>>>> The Time Now is => 60178.432 Periodic Utilization Budget => 5.00E-01 <<<<<<<<<

Task ID	Durations Mandatory	Optional	Time Remaining	Start-Time	Deadline	Started-At	Importance	Status	Latest Start-Time
749552	0.100	1.000	0.749	60175.363	60179.886	N/A	2	5	60179.137

761232	0.100	1.000	2.000	60175.674	60186.718	N/A	3	5	60184.718
772912	0.100	1.000	2.000	60175.983	60186.558	N/A	4	5	60184.558
784592	0.100	1.000	2.000	60176.392	60189.011	N/A	5	5	60187.011
796272	0.100	1.000	2.000	60176.713	60187.154	N/A	6	5	60185.154
807952	0.100	1.000	0.926	60177.015	60179.735	N/A	7	5	60178.809
819632	0.100	1.000	2.000	60177.374	60181.281	N/A	8	5	60179.281
831312	0.100	1.000	2.000	60177.651	60189.236	N/A	9	5	60187.236
842992	0.100	1.000	2.000	60177.971	60184.258	N/A	10	5	60182.258
854672	0.100	1.000	0.146	60178.301	60183.336	60178.324	11	2	

>>>>>>>>> The Time Now is => 60180.039 Periodic Utilization Budget => 5.00E-01 <<<<<<<<<

	Durations		Time						Latest
Task ID	Mandatory	Optional	Remaining	Start-Time	Deadline	Started-At	Importance	Status	Start-Time
749552	0.100	1.000	0.749	60175.363	60179.886	N/A	2	6	
761232	0.100	1.000	2.000	60175.674	60186.718	N/A	3	6	
772912	0.100	1.000	2.000	60175.983	60186.558	N/A	4	6	
784592	0.100	1.000	2.000	60176.392	60189.011	N/A	5	6	
796272	0.100	1.000	2.000	60176.713	60187.154	N/A	6	6	
807952	0.100	1.000	0.100	60177.015	60179.735	N/A	7	4	60179.635
819632	0.100	1.000	2.000	60177.374	60181.281	N/A	8	6	
831312	0.100	1.000	2.000	60177.651	60189.236	N/A	9	6	
842992	0.100	1.000	2.000	60177.971	60184.258	N/A	10	6	
854672	0.100	1.000	2.000	60178.301	60183.336	N/A	11	6	
878032	0.100	1.000	0.200	60179.983	60184.180	60180.036	1	2	

>>>>>>>>> The Time Now is => 60180.101 Periodic Utilization Budget => 5.00E-01 <<<<<<<<<

	Durations		Time						Latest
Task ID	Mandatory	Optional	Remaining	Start-Time	Deadline	Started-At	Importance	Status	Start-Time
749552	0.100	1.000	0.749	60175.363	60179.886	N/A	2	6	
761232	0.100	1.000	2.000	60175.674	60186.718	N/A	3	6	
772912	0.100	1.000	2.000	60175.983	60186.558	N/A	4	6	
784592	0.100	1.000	2.000	60176.392	60189.011	N/A	5	6	
796272	0.100	1.000	2.000	60176.713	60187.154	N/A	6	6	
807952	0.100	1.000	0.100	60177.015	60179.735	60180.099	7	2	
819632	0.100	1.000	2.000	60177.374	60181.281	N/A	8	6	
831312	0.100	1.000	2.000	60177.651	60189.236	N/A	9	6	
842992	0.100	1.000	2.000	60177.971	60184.258	N/A	10	6	
854672	0.100	1.000	2.000	60178.301	60183.336	N/A	11	6	
878032	0.100	1.000	0.166	60179.983	60184.180	N/A	1	6	

Appendix B. Periodic Priority Assignment Methods Investigated

Using rate monotonic theory to schedule the periodic tasks requires that priorities be assigned based upon the period of each task. In theory, this sounds simple, but in practice, additional problems arise. In particular, a problem arises when there are more periodic tasks than there are periodic priorities. The investigation undertaken to solve that problem is outlined in the following sections.

B.1. Periodic Priority Assignment Problem and Potential Solution Methods

The problem of assigning priorities to periodic tasks is compounded by the dynamic nature of the system. At any one time there may be only a handful of tasks, with more than enough distinct priorities to assign a unique priority to each task. At other times, there may be many more periodic tasks than periodic priorities. When this condition occurs, the problem becomes one of determining which tasks to assign to which priority 'bin'. The problem is better illustrated in Figure B.1.

By examining the figure, four possible groupings for the tasks stand out. This tends to put a large number of tasks into the priority bins for tasks of shorter periods and hardly any in the priority bins for the tasks with longer periods. A solution might be to assign tasks based upon some predetermined method or try to calculate the best method for the current situation. It is possible for the situation shown above to be reversed, and have a large number of tasks with long periods and only a small number of tasks with short periods. Four methods were developed to handle the problem under different conditions. No attempt is made to determine which is the best, rather this research illustrates that different solutions are possible and should be examined in future work. The four methods developed are called Static, Linear, Normal Distribution, and Simple.

The simple method is for the trivial case were the number of periodic tasks is less than the number of periodic priorities. A count of the number of currently active periodic tasks is always maintained by the system and the simple method is always employed if it can be. This method meets the theoretical basis used in the rate monotonic algorithm and assigns each periodic task a distinct priority and is the ideal method of assigning priorities when using the rate monotonic algorithm. The other methods are described in the following sections.

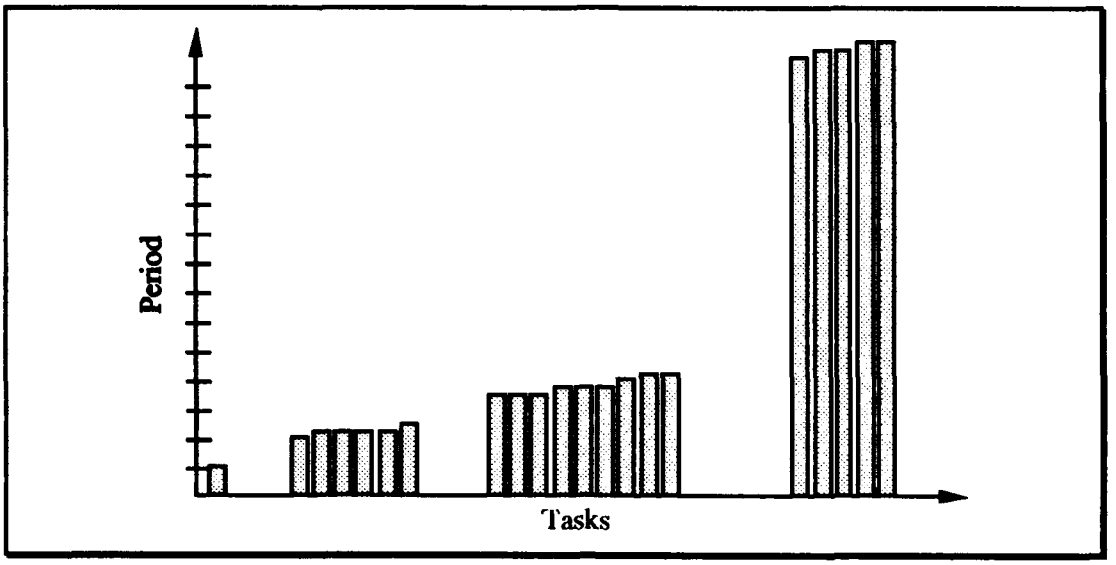

Figure B.1 Example Periods of a Task Set

B.1.1. Periodic Priorities Normal Distribution Method. The Normal Distribution method is designed around the assumption that the periodic task's periods will, as the name implies, have a statistical normal distribution. Thus, the middle priorities have more tasks than either the highest or lowest priorities. There is no empirical basis for this assumption and the purpose of implementing it is to show how statistical distributions can be used to help assign priorities in a dynamic system. Other distributions are likely in any given domain. Figure B.2 illustrates the concept.

The algorithm used to implement the normal distribution method uses the standard formula's to calculate the mean and variance [Allen, 1990]. The mean and standard deviation can be quickly calculated with each new task that is added or removed. Once they are calculated, the values of the periods that belong to each priority bin are calculated by dividing the spread between ± 2 standard deviations from the mean by the number of periodic priorities minus two. The highest periodic priority is used for tasks with periods that are less than 2 standard deviations below the mean period, and the lowest periodic priority is used for tasks with periods that are greater than 2 standard deviations above the mean period.

B.1.2. Periodic Priorities Linear Methor. The Linear Method of assigning periodic priorities is illustrated in Figure B.3. The basic idea is to assign period ranges to the different priorities using a straight line equation generated by using the task with the longest period. Note that the periodic task with the longest period is simply the task at the tail of the Tasks_By_Period_Queue.

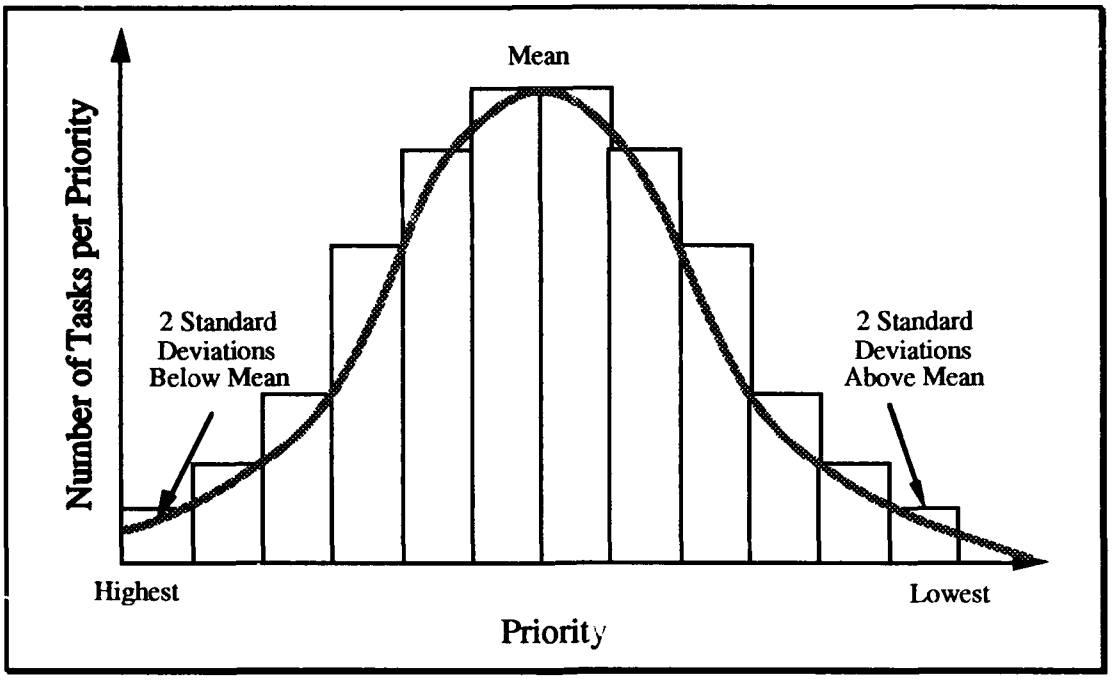

Figure B.2 Periodic Priorities Using Normal Distribution

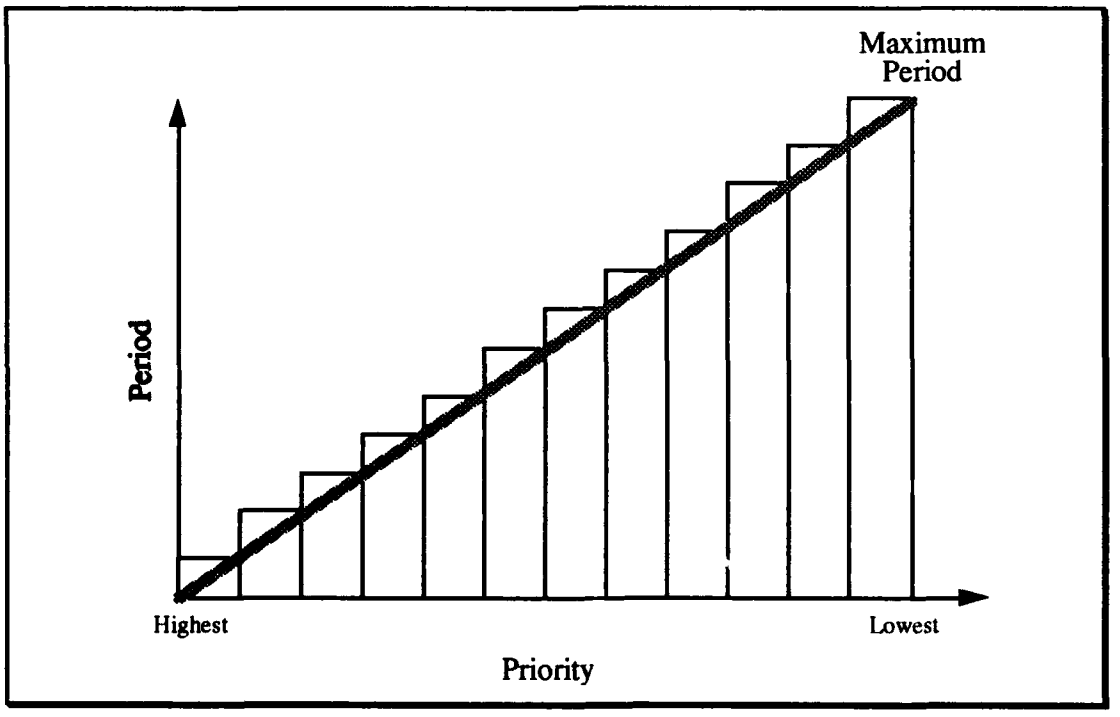

Figure B.3 Periodic Priorities Using Linear Method

The slope of the line generated by starting at the origin and proceeding to the maximum period (refer to Figure B.3), is simply equal to the maximum period divided by the number of periodic priorities. Once this slope is known, calculating the range of periods associated with each priority is a matter of plugging each priority into the straight line equation $y = mx$, where x is the priority and m is the slope calculated above and assigning the result as the maximum period for any task with that priority.

Unlike the Normal Distribution method, this method assumes that task periods is evenly distributed. The result of using this method is to split the tasks into the different priority bins so that the higher priority bins have fewer tasks than the lower priority bins.

B.1.3. Periodic Priorities Static Method. The Static method differs from the other two in that the spread of periods assigned to each priority is defined before the system starts executing and does not change while the system is executing. Figure B.4 below illustrates the Static Method. The static method is clearly the most efficient, both in its memory requirements and its execution speed. The tradeoff, of course, is dealing with widely varying task sets with widely varying periods.

To determine which method is best, an analysis of the results of each method will have to be made. It is important to remember that the problem these methods are designed to handle is more tasks than priorities. The initial guess is that given some random task set, the static method has the least distribution of the tasks over the available priorities while the linear method has the best.

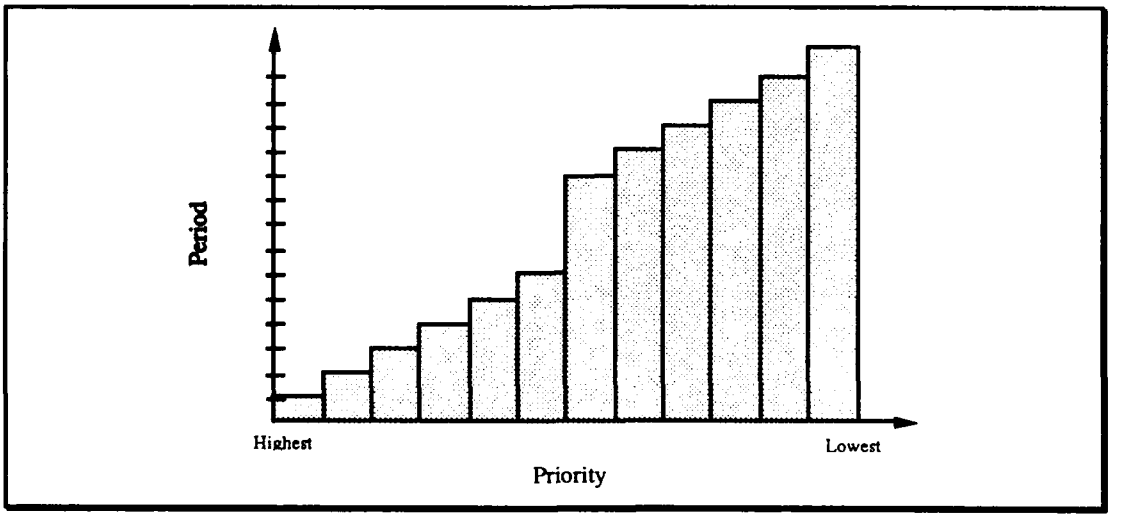

Figure B.4 Periodic Priorities Using Static Method

B.2. Evaluation of Priority Assignment Methods

Perhaps the area of greatest misdirected effort was in the thought and development of the differing methods to assign periodic priorities. The assumption was that assigning priorities to insure that the rate monotonic theory held would be difficult when the number of tasks, exceeded the number of priorities. Complicating the issue was the belief the unknown spread of periods might make it difficult to spread tasks effectively over the available periods. This is indeed a problem if the periods for each priority are assigned statically, but as was discovered, can be simply done dynamically.

Before the discussion of the results of testing the various methods, a note about the graphs included in this section is necessary. First, there are two types of graphs included. The first type shows the period of a task versus its assigned priority. The second graph types shows the number of tasks assigned to each priority. In addition, there are four graphs for each priority assignment method; static, linear, normal distribution, and a new method developed during this testing. One set of two of the graphs for each method reflects the condition when both the mandatory part and the optional part of all the periodic tasks has been scheduled (i.e. a non-overloaded condition). The other set of two reflects the condition when only the mandatory part of some periodic tasks are scheduled (i.e. an overload condition). Graphs showing the middle condition, where some tasks are scheduled to execute both their mandatory and optional parts, and some only are scheduled to execute their mandatory parts, are not included because the priority assignments are identical to that of the all optional condition. The difference lies in the execution mode assigned each task, not the priorities.

As mentioned above, results show a lot of unnecessary effort was put into this area. The belief that a serious problem may develop if priority bins are defined statically is, however, well founded. Figure 6.2 thru Figure 6.5 clearly show the problem. Because, the size of each priority bin has been defined statically, it obviously does not handle a dynamic set of tasks well. Note that in both Figure 6.3 and Figure 6.5 only a small number of the available priorities are used. This can potentially lead to a significant problem when using rate monotonic theory, essentially lowering the utilization levels needed to insure the tasks will complete by their deadlines [Sha, 1989].

There are some factors that can be used to improve upon the static method. First, for this thesis, periods are generated using a random number generator. In any "real" application, periods would be assigned based upon some definable criteria. It is clearly possible to better match the static priorities to the

particular system. Note, however, that at any instant in time, it is still possible to have a very uneven distribution of tasks to priorities. I would suggest abandoning static priority assignments altogether for any continuation of the work in this thesis.

Figure B.9 thru Figure B.16 show a sample of the test results of the Linear and Normal Distribution methods developed. In Figure B.9, Figure B.11, Figure B.13, and Figure B.15, the slope of the line shows that indeed priorities are being assigned correctly (i.e. rate monotonically). However, Figure B.10, Figure B.12, Figure B.14, and Figure B.16 show that the distribution of tasks to priorities, although much better than the static method, is very uneven. The results of these distribution methods on run time performance can not be accurately ascertained with the current implementation, but from the prospective of effective utilization of available priorities, this approach is unacceptable.

Analysis of the graphs in Figure B.9 thru Figure B.16 lead to the development of a much simpler priority assignment method. The "new" method simply takes the number of tasks and divides by the number of priorities. The integer result of that division is then used to assign that number of tasks to each bin. Any remaining tasks are assigned the lowest periodic priority. This approach eliminated the need to maintain an array that held the allowable values for each priority given the current task set. In addition, the calculation of that array was eliminated. Thus, the "new' method is simpler, more time efficient and more space efficient. The results from a run using the new method are shown in Figure B.17 thru Figure B.20.

Note that there is problem still in the implementation when an overload condition occurs as shown by the spread of priorities in Figure B.20. The problem stems from using the fact that the routine to assign the priorities uses the total number of periodic tasks currently in the system, instead of the number of periodic tasks in the queue it is passed to assign priorities to. The fix is relatively simple and will be implemented in future versions.

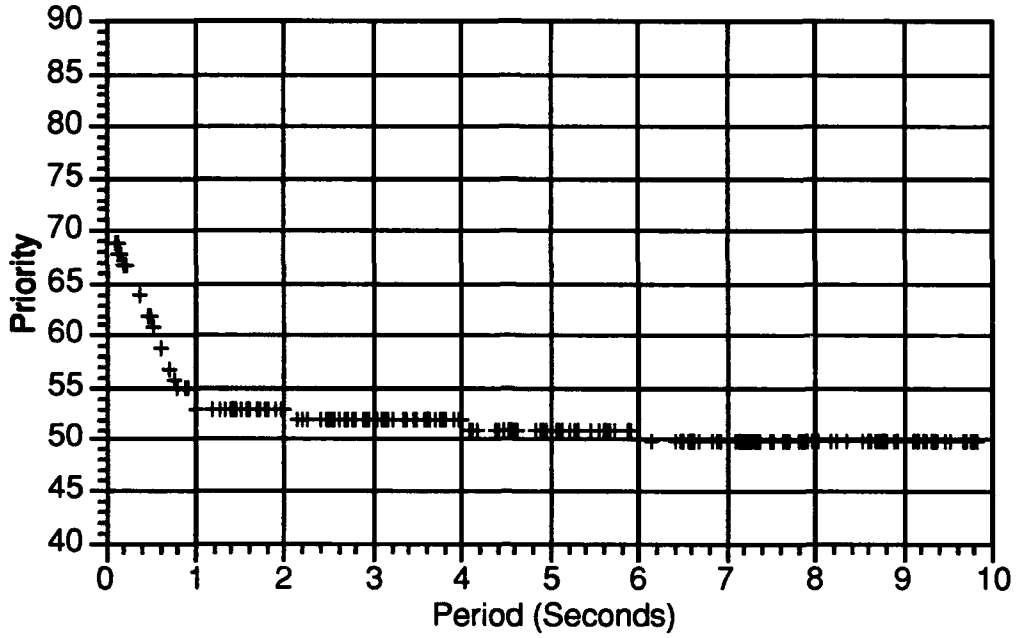

Figure B.5 Period vs. Priority, Static Method, All Optional

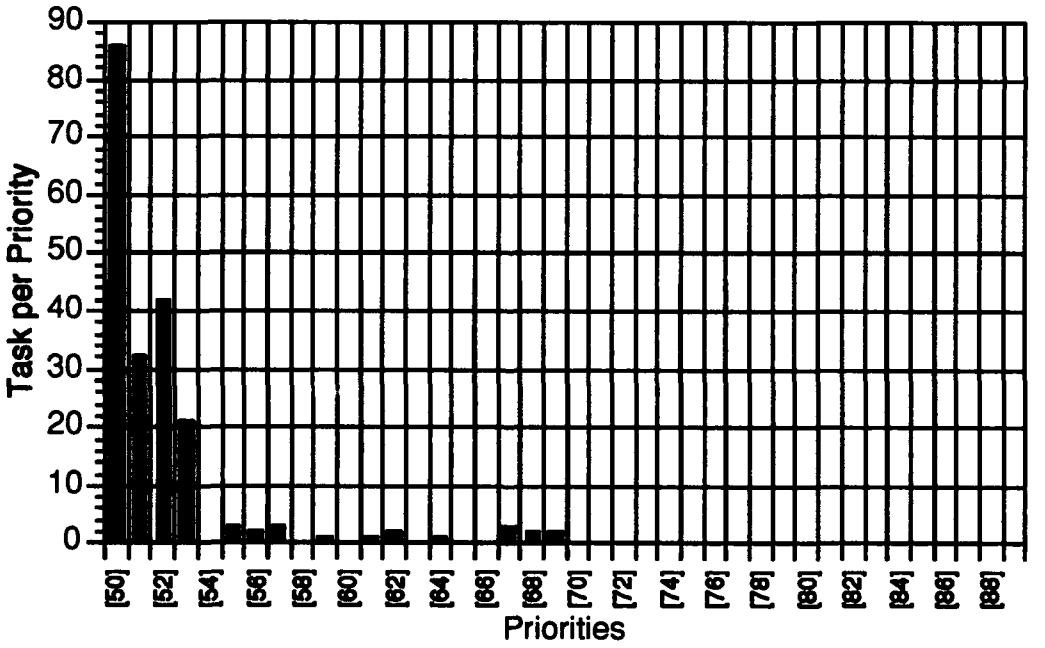

Figure B.6 Tasks per Priority, Static Method, All Optional

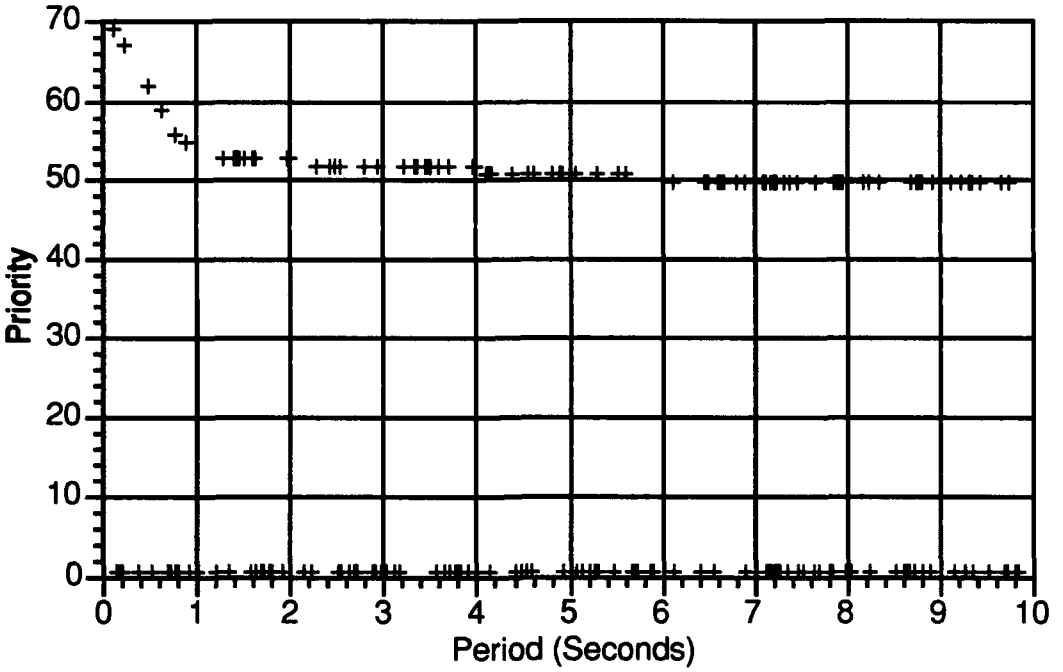

Figure B.7 Period vs. Priority, Static Method, Some Mandatory

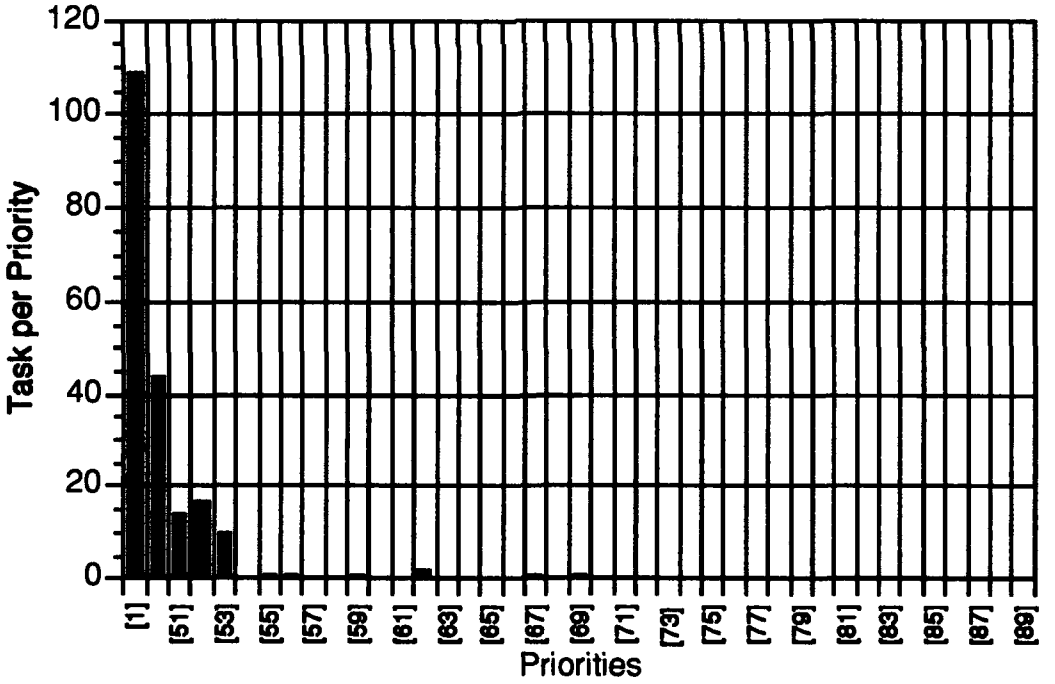

Figure B.8 Tasks per Priority, Static Method, Some Mandatory

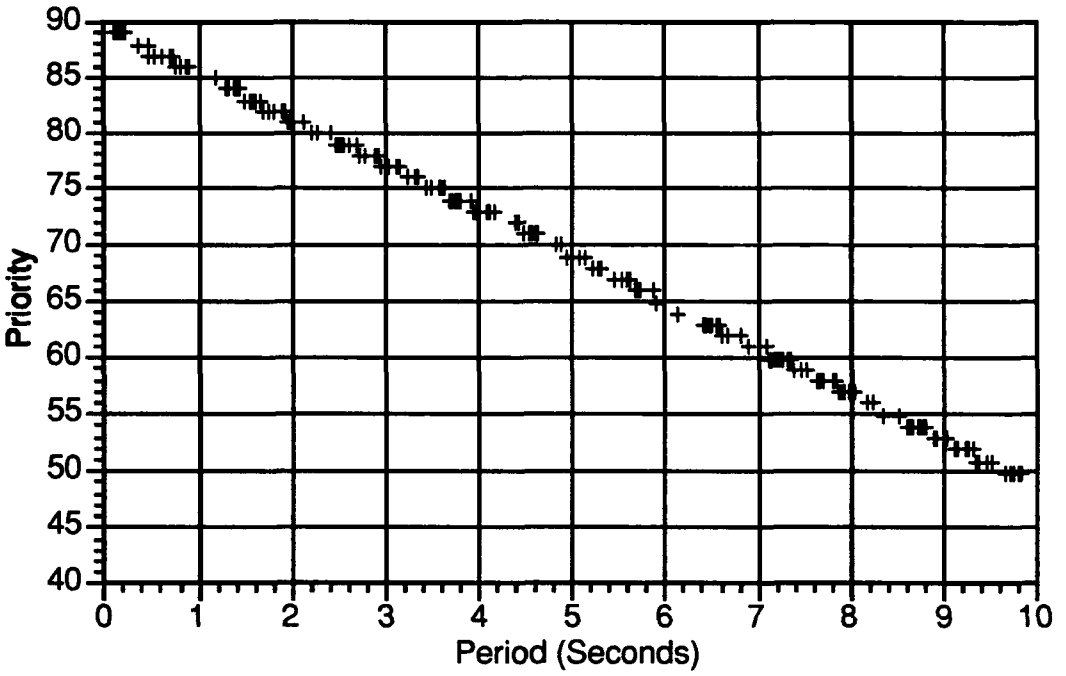

Figure B.9 Period vs. Priority, Linear Method, All Optional

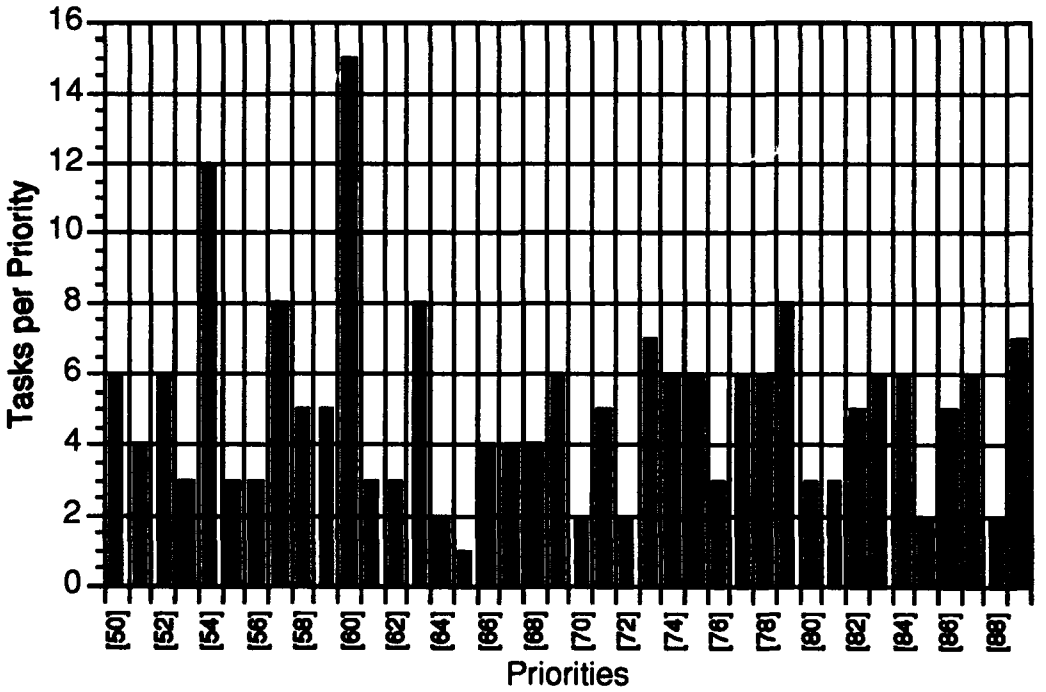

Figure B.10 Tasks per Priority, Linear Method, All Optional

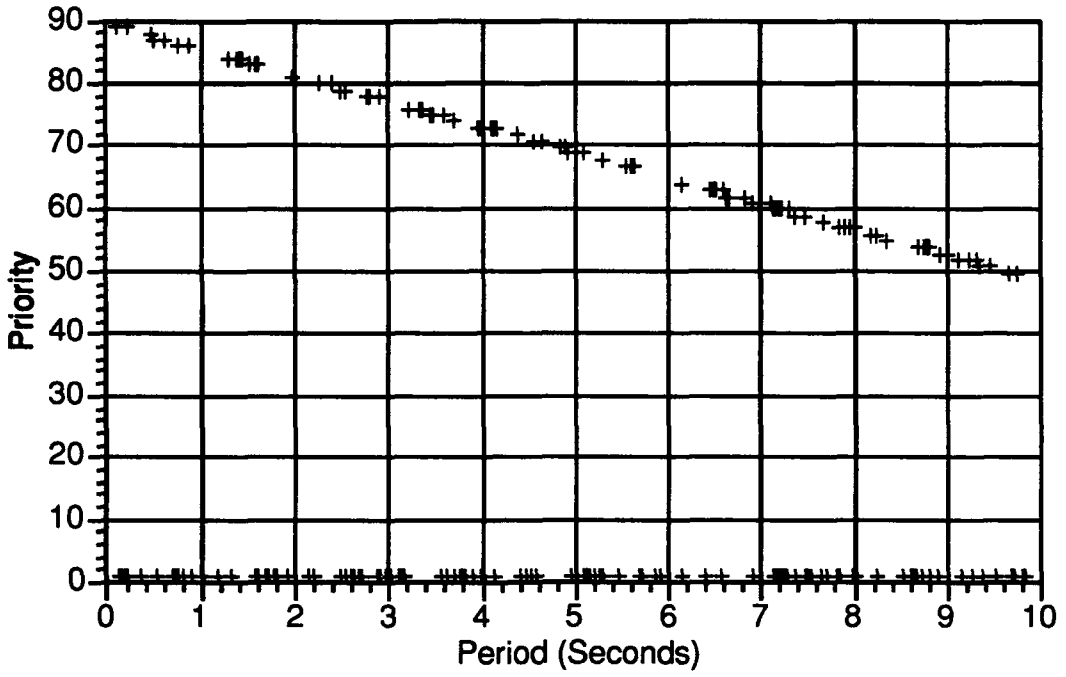

Figure B.11 Period vs. Priority, Linear Method, Some Mandatory

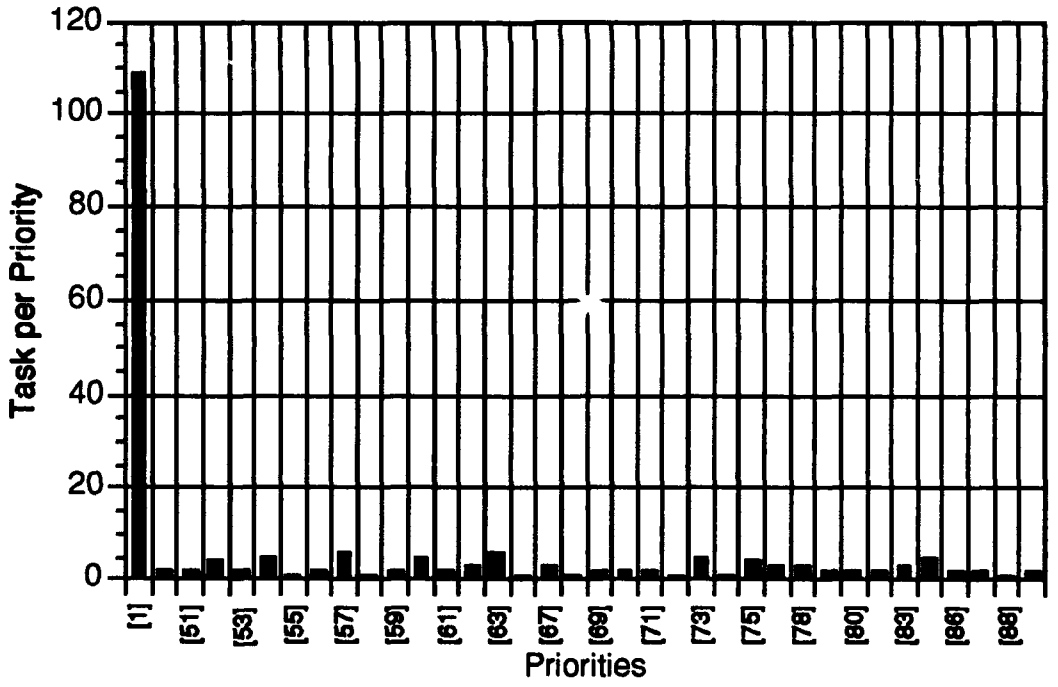

Figure B.12 Tasks per Priority, Linear Method, Some Mandatory

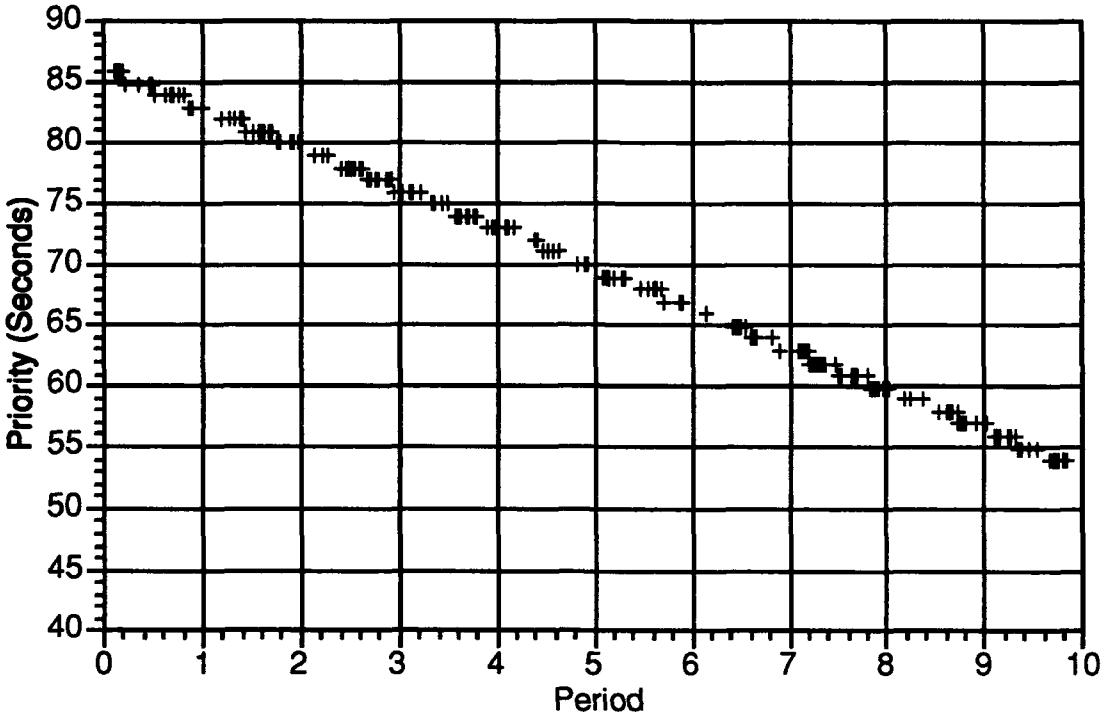

Figure B.13 Period vs. Priority, Normal Distribution Method, All Optional

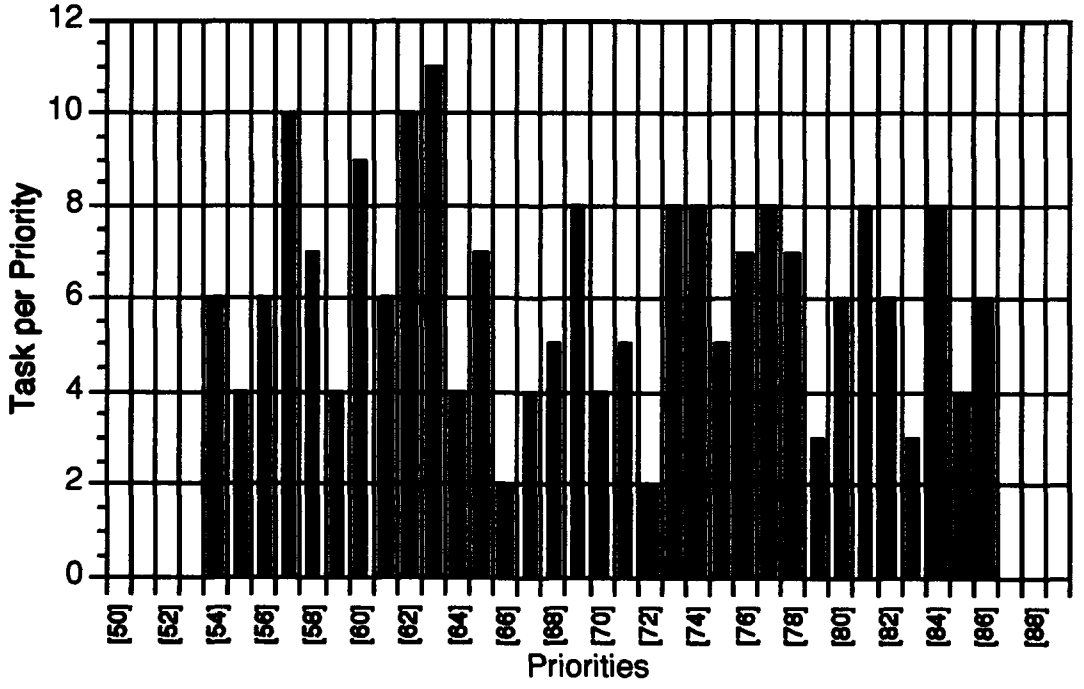

Figure B.14 Tasks per Priority, Normal Distribution Method, All Optional

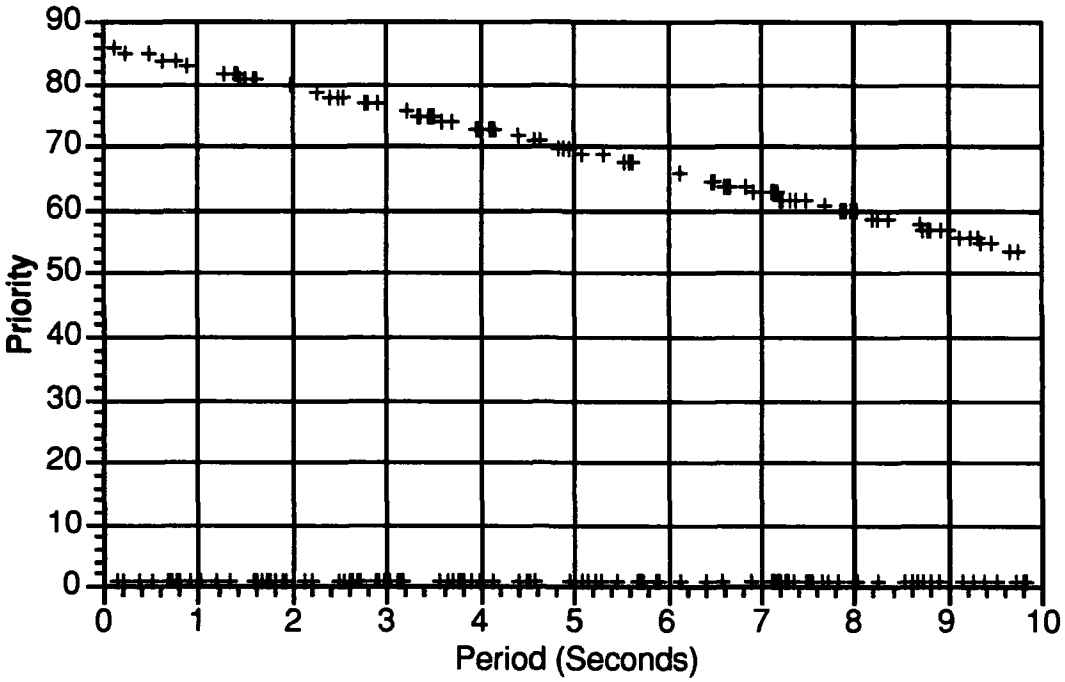

Figure B.15 Period vs. Priority, Normal Distribution Method, Some Mandatory

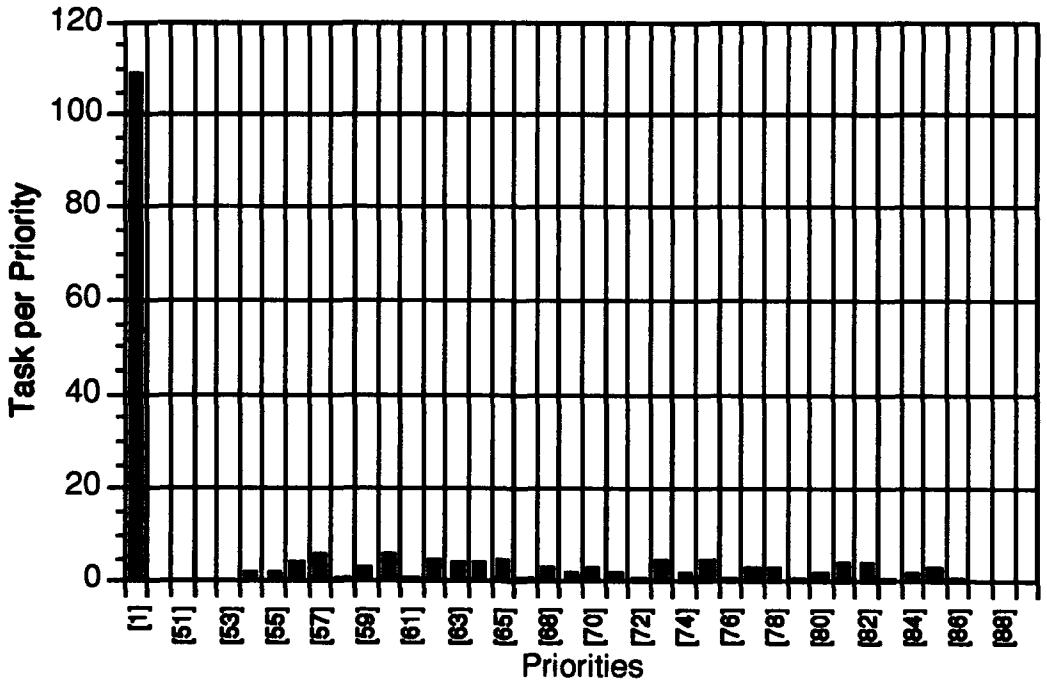

Figure B.16 Tasks per Priority, Normal Distribution Method, Some Mandatory

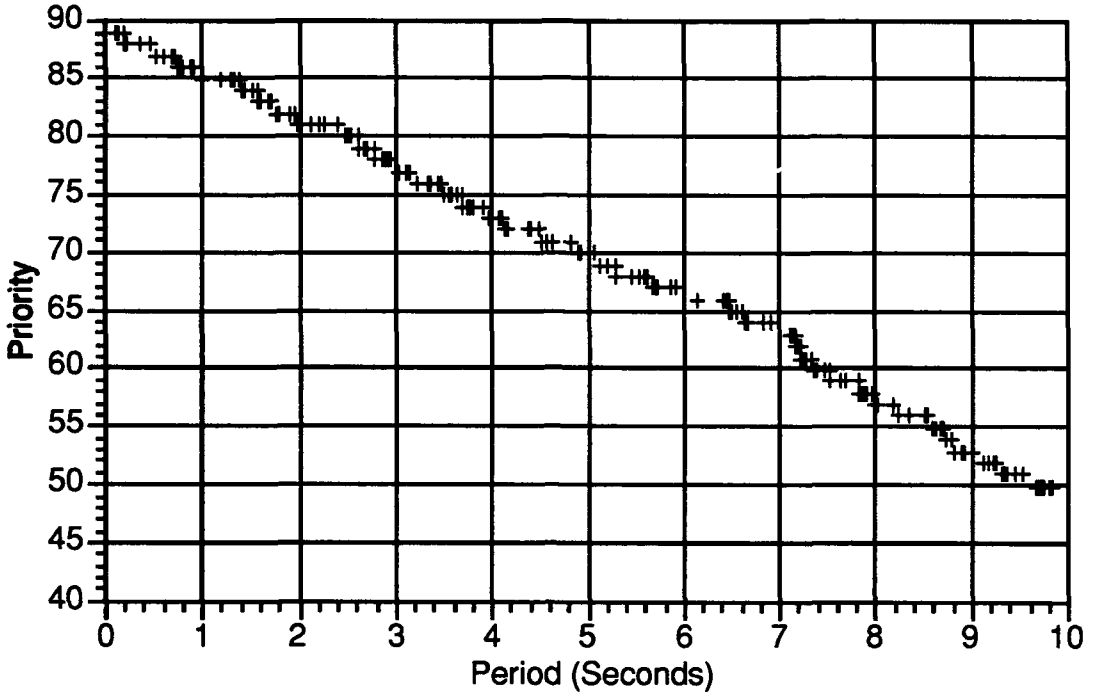

Figure B.17 Period vs. Priority, New Method, All Optional

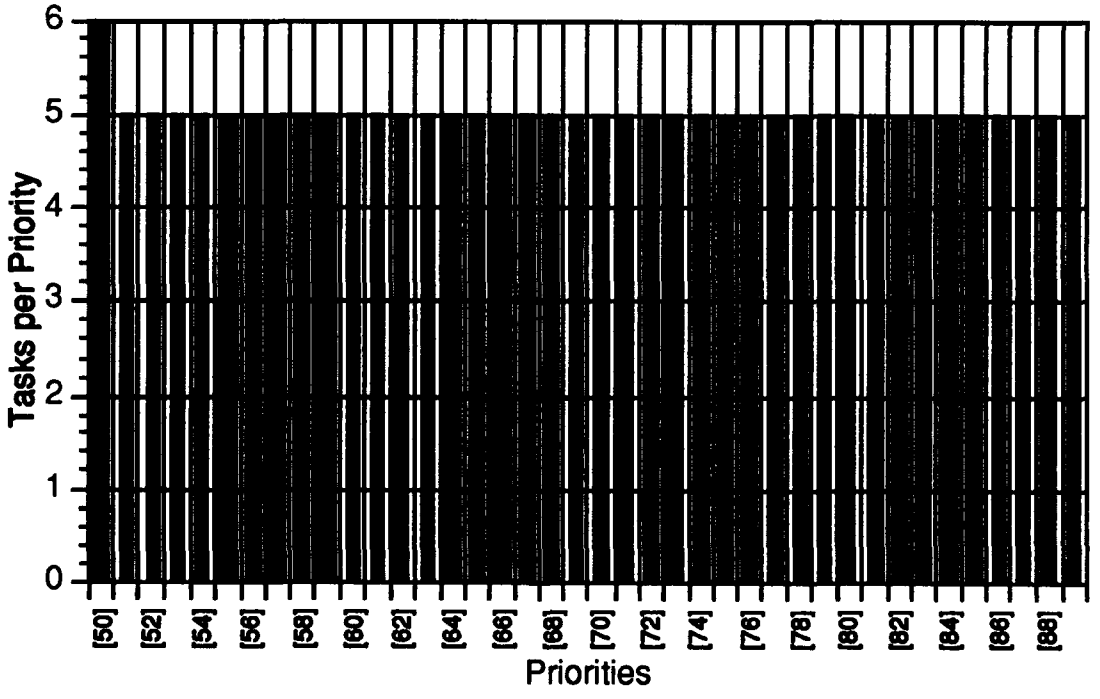

Figure B.18 Tasks per Priority, New Method, All Optional

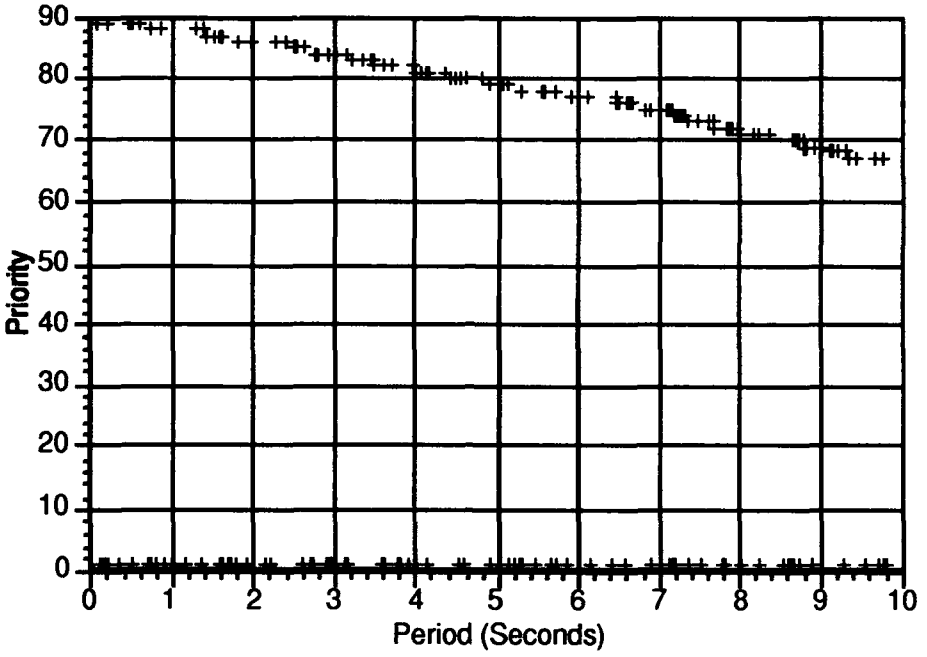

Figure B.19 Period vs. Priority, New Method, Some Mandatory

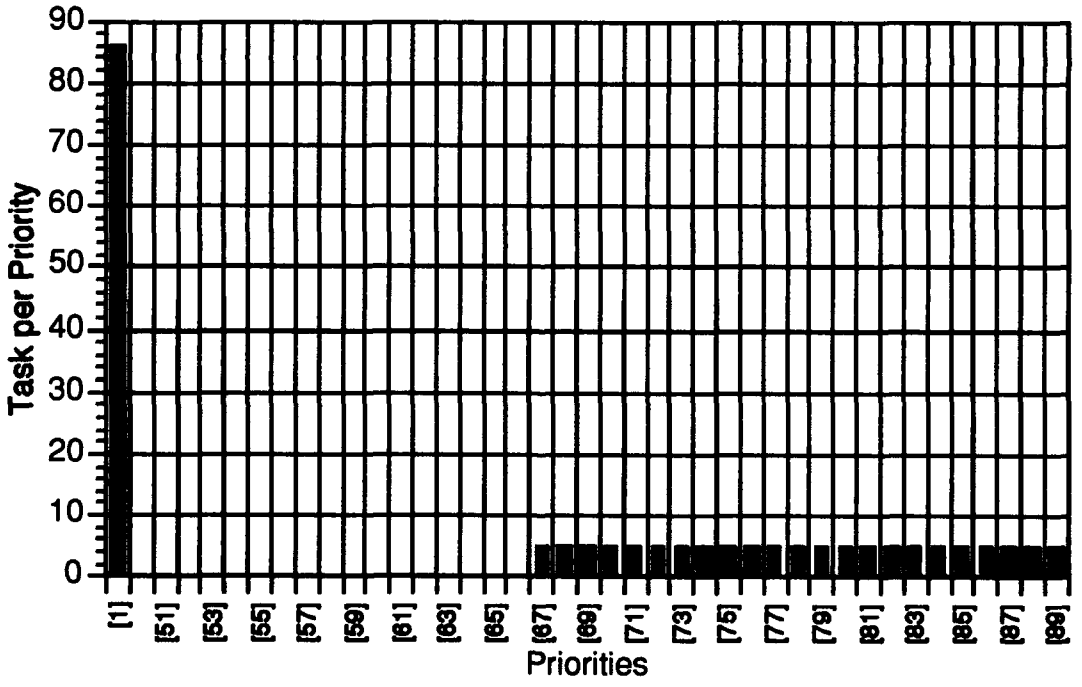

Figure B.20 Tasks per Priority, New Method, Some Mandatory

Appendix C. IRTS Demonstration System User's Guide

This appendix explains how to compile and use the IRTS feasibility demonstration code. The first section details the system requirements, recommended library structure, and compilation order. The second section contains the source for the feasibility demonstration system. The source code is included for reference only and is not intended to be "production quality code". Chapter 6 of this thesis has pointed out a number of deficiencies and recommended the changing some of the data structures used. Any future work should address these issues before continuing to use the demonstration code.

C.1 System Requirements and Compilation Order

The IRTS demonstration system requires Verdix Ada 6.0 in order to compile. It uses the Verdix supplied task management procedures Suspend_Task, Resume_Task, and Set_Priority. To use these features, it is necessary to link in the Verdix library V_XTasking. In addition, CLIPS/Ada is used as the Reasoning Process and also must be compiled and linked to the users library. Also, the Booch component files Storage_Manager_Sequential, Deque_Priority_Balking_Sequential_Unbounded_Managed_Iterator, and Monitor are required. Figure C.1 shows the recommend directory structure.

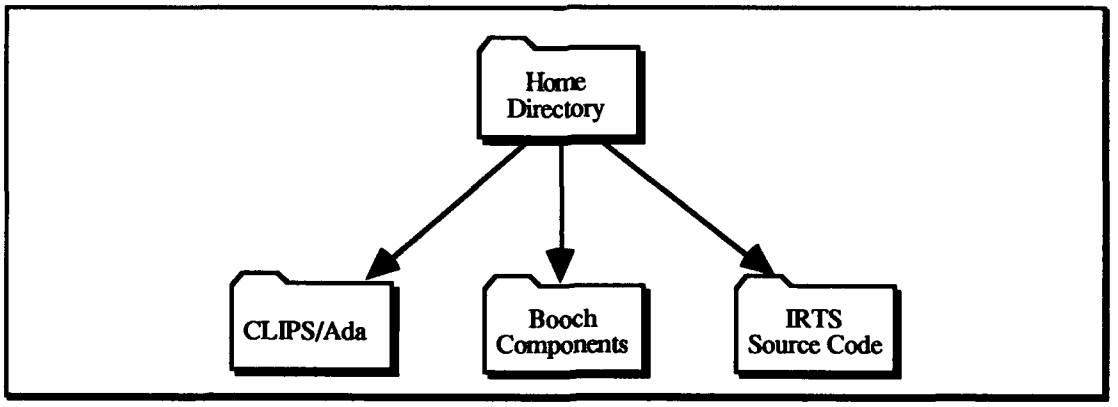

Figure C.1 Recommended Directory Structure for IRTS Demonstration System

After creating a new Verdix Ada Library in the IRTS Source Code directory, modify the file "ada.lib" to reflect the locations of CLIPS/Ada and the required Booch components. In addition, the

"vads_exec" library should be linked in. An example ada.lib file after these modifications should look as follows:

```
!ada library
ADAPATH= /usr/vads6_0/verdixlib /usr/vads6_0/standard /usr/vads6_0/vads_exec
ADAPATH= /home/hawkeye3/mawhelan/Booch /home/hawkeye3/mawhelan/CLIPS_Ada
```

Note that the third line reflects user specific locations for the CLIPS/Ada source code and Booch components. These locations should reflect the actual locations for the users directory structure.

To compile the IRTS demonstration system, first compile the CLIPS/Ada source code (refer to the CLIPS/Ada manual) and the required Booch components. After those files have been successfully compiled, the IRTS files should be compiled in the following order:

random.a
global_data_types_spec.a
support_functions.a
task_control_buffer_spec.a
periodic_tasks_spec.a
any_time_tasks_spec.a
singular_tasks_spec.a
task_manager_spec.a
reasoning_process_spec.a
io_process_spec.a
task_control_buffer_body.a
periodic_tasks_body.a
any_time_tasks_body.a
singular_tasks_body.a
task_manager_body.a
userfun_body.a
reasoning_process_body.a
io_process_body.a
task_manager.a
get_tcb.a
find_tcb.a
feasible.a
p_tasks_<_p_priorities.a
p_tasks_>_p_priorities.a
pp_by_importance.a
some_periodics_optional.a
assign_periodic_priorities.a
schedule.a
un_schedule.a
non_periodic_completed.a
modify.a
modified_feasible.a
dispatch_tasks.a
print_periodic_tasks.a
print_non_periodic_tasks.a
print_test_times.a
record_times.a

The IRTS demonstration system makes use of a file called "test_rp_rules.clp". The file contains the CLIPS/Ada rules that drive the Reasoning Process. In addition to all the functions available in CLIPS/Ada,

a number of additional functions used to communicate with the Task Manager are available. The Task Manager functions are as follows (all times are given as a real number of seconds from midnight):

- *(add_task <display flag> <task type> <procedure id> <period> <deadline> <starttime> <importance>)* The *display flag* is a boolean value that instructs the task to print out its start and stop times while executing. *task type* is either any-time, periodic, or singular. *procedure id* is the identifier of the procedure within the *task type* package that this task shell is to use. *period* is the value to be used as the period. The use of the *deadline* value depends on the *task type*. If the *task type* is periodic, then the *deadline* is the stop time for the periodic task; otherwise, the *deadline* is the time by which a task must provide an answer. *start time* is the time when the task is eligible to start execution. *importance* is the task's global relevance in relation to all other tasks. The procedure returns an integer value that represents the task's id. It must be used to refer to the task whenever the modify or remove procedures are used.

- *(modify_task <display flag> <period> <deadline> <starttime> <importance> <task id>)* The parameters should reflect the new values desired. The old values should be used for those parameters that are not to be changed.

- *(remove_task <task id>)* This procedure simply removes the specified task from the currently active set.

- *(new_periodic_utilization <value>) value* is the new amount of processor utilization to allot for periodic tasks. It should range between 0.0 and 1.0.

In addition to these procedures, the Task Manager also communicates with the Reasoning Process by asserting facts into the CLIPS/Ada fact base. The Task Manager signals infeasible tasks, missed deadlines, completed tasks, and its current status through facts. The facts asserted by the Task Manager have the following forms:

- *(infeasible_task <task id>)* This fact is asserted whenever a modify or add operation has tried to assert an infeasible task as described in Chapter 5.

- *(missed_deadline <task id> <time>)* This fact is asserted whenever a periodic task does not complete its execution during its period or a non-periodic task fails to complete its execution by its deadline. *time* is the time at which the missed deadline was detected.

- *(task_manager_status <time> <PU> <MU> <OU> <RU> <periodic condition> <#periodic tasks> <#non-periodic tasks>) time* is the time at which this status report was generated. *PU* is the current periodic utilization. *MU* is the utilization of the mandatory parts of the current periodic task set. *OU* is the utilization of the optional parts of the current periodic task set. *RU* is the utilization required given the current number of periodic tasks and the current budgeted utilization. *periodic condition* is either SOME_MANDATORY, SOME_OPTIONAL, or ALL_OPTIONAL as described in Chapter 5. *#periodic tasks* and *#non-periodic tasks* are the number of currently active tasks of each type. A status report is sent after an Add_Task, Modify_Task, Remove_Task, or Change_Periodic_Utilization entry is called.

C.2 Source Code

The source code developed for this research effort follows. The purpose of its inclusion is to allow future researchers to see a concrete example of how the architecture outlined in this thesis has been implemented. No attempt has been made to make the code robust or tuned for optimal performance and no such claims are made. The source code should be used as a guide for any future research efforts and not as a fully developed Intelligent Real-Time System.

```
--*****************************************************************************--
--*                                                                           *--
--*                    Intelligent Real-Time System                           *--
--*                                                                           *--
--*****************************************************************************--
--*                                                                           *--
--*   TITLE: IRTS                                                             *--
--*                                                                           *--
--*   DATE: 1 December 1992                                                   *--
--*                                                                           *--
--*   VERSION: 1.0                                                            *--
--*                                                                           *--
--*   AUTHOR: Michael A. Whelan                                               *--
--*                                                                           *--
--*   LANGUAGE: Verdix Ada Version 6.0                                        *--
--*                                                                           *--
--*****************************************************************************--
with Text_IO,
          IO_Process_Package,
          Reasoning_Process_Package,
          Task_Manager_Package ;

procedure IRTS is
pragma Priority ( 99 ) ;
begin
    Reasoning_Process_Package.Reasoning_Process.Initialize ;
    IO_Process_Package.IO_Process.Initialize ;
    Task_Manager_Package.Task_Manager.Initialize ;
end IRTS;
```

```
--*****************************************************************************--
--*                                                                           *--
--*                    Package Global_Data_Types                              *--
--*                                                                           *--
--*****************************************************************************--
package Global_Data_Types is

    type Mode_Type is ( Mandatory, Optional ) ;

    --
    -- This enumeration type needs to have the name of every task that is
    -- available for the system to create.
    --
    type Task_Kind_Type is ( PERIODIC, ANY_TIME, SINGULAR ) ;

    type Periodic_Condition_Type is ( SOME_MANDATORY, SOME_OPTIONAL,
                                      ALL_OPTIONAL ) ;
    type Event_Type is ( PERIODIC_EVENT, ANY_TIME_EVENT, SINGULAR_EVENT ) ;

end Global_Data_Types ;
```

```
--*****************************************************************************--
--*                                                                           *--
--*                          Random_Numbers                                   *--
--*                                                                           *--
--*****************************************************************************--
package random_numbers is

-- adapted from "Random number generators: good ones are hard to find"
--              CACM, vol 31 #10 (Oct 88) pp. 1192-1201
-- the algorithm itself is given on page 1196

   function random return float;

   seed: float :=  3.141592653589793; -- or this can be set by the
                                      -- user

end random_numbers;
package body random_numbers is

   function random return float is
      a: constant float := 16807.0;
      m: constant float := 2147483647.0;
      q: constant float := 127773.0;     --  m div a
      r: constant float := 2836.0;       --  m mod a
      lo, hi, test: float;
   begin
      hi := float( integer( seed / q ) );
      lo := seed - q * hi;
      test := a * lo - r * hi;
      if test > 0.0 then
         seed := test;
      else
         seed := test + m;
      end if;
      return seed / m;
   end random;
end random_numbers;
```

```
--*****************************************************************************--
--*                                                                           *--
--*                      Any_Time_Tasks Package Spec                          *--
--*                                                                           *--
--*****************************************************************************--
--*                                                                           *--
--*   TITLE: Any_Time_Tasks                                                   *--
--*                                                                           *--
--*   DATE: 1 December 1992                                                   *--
--*                                                                           *--
--*   VERSION: 1.0                                                            *--
--*                                                                           *--
--*   AUTHOR: Michael A. Whelan                                               *--
--*                                                                           *--
--*   LANGUAGE: Verdix Ada Version 6.0                                        *--
--*                                                                           *--
--*****************************************************************************--
with System,
     Global_Data_Types ;
use  System,
     Global_Data_Types ;

package Any_Time_Tasks is

    ------------------------------------------------------------
    --                                                        --
    --                  TYPE DECLARATIONS                     --
    --                                                        --
    ------------------------------------------------------------
    subtype Any_Time_Types is Integer range 1..10 ;
    type Any_Time_Variables_Type is
        record
            Kind     : Integer ;
            Display  : Boolean := FALSE ;
            Continue : Boolean := TRUE ;
            Mode     : Global_Data_Types.Mode_Type ;
        end record ;

    ------------------------------------------------------------
    --                                                        --
    --     TASK TYPE Generic_Any_Time DECLARATION             --
    --                                                        --
    ------------------------------------------------------------
    task type Generic_Any_Time is
        entry Initialize ( The_Variables : in  Any_Time_Variables_Type ;
                           The_Task_ID   : out System.Task_ID ;
                           Man_Duration  : out Duration ;
                           Opt_Duration  : out Duration ) ;
        entry Change_Variables ( The_Variables : in Any_Time_Variables_Type ;
                                 Man_Duration  : out Duration ;
                                 Opt_Duration  : out Duration ) ;
    end Generic_Any_Time ;

    type Any_Time_Task_Ptr is access Generic_Any_Time ;

    procedure Create  ( The_Variables : in  Any_Time_Variables_Type ;
                        The_Task_Ptr  : out Any_Time_Task_Ptr ;
                        The_Task_ID   : out System.Task_ID ;
                        Man_Duration  : out Duration ;
                        Opt_Duration  : out Duration ) ;

    procedure Store_Variables ( The_Variables : in Any_Time_Variables_Type ;
                                The_Task_ID   : in System.Task_ID ) ;

    procedure Remove_Variables ( The_Task_ID : in System.Task_ID ) ;

end Any_Time_Tasks ;
```

```
--*****************************************************************************--
--*                                                                           *--
--*                    Any_Time_Tasks Package Body                            *--
--*                                                                           *--
--*****************************************************************************--
--*                                                                           *--
--*   TITLE: Any_Time_Tasks                                                   *--
--*                                                                           *--
--*   DATE: 1 December 1992                                                   *--
--*                                                                           *--
--*   VERSION: 1.0                                                            *--
--*                                                                           *--
--*   AUTHOR: Michael A. Whelan                                               *--
--*                                                                           *--
--*   LANGUAGE: Verdix Ada Version 6.0                                        *--
--*                                                                           *--
--*****************************************************************************--
with Calendar,
     Support_Functions,
     Task_Control_Buffer,
     Task_Manager_Package,
     V_XTasking ;
use  Calendar,
     Support_Functions,
     Task_Manager_Package,
     V_XTasking ;

package body Any_Time_Tasks is

    ------------------------------------------------------------
    --                                                        --
    --                  TYPE DECLARATIONS                     --
    --                                                        --
    ------------------------------------------------------------
    type Durations_Type is array ( Global_Data_Types.Mode_Type,
                                   Any_Time_Types ) of Duration ;

    package Any_Time_Variables_Buffer is new Task_Control_Buffer
                                        ( Item  => Any_Time_Variables_Type,
                                          Index => System.Task_ID ) ;

    ------------------------------------------------------------
    --                                                        --
    --                  OBJECT DECLARATIONS                   --
    --                                                        --
    ------------------------------------------------------------
    Durations : Durations_Type := ( Mandatory => ( 1  => ( 0.1 ),
                                                   2  => ( 0.002 ),
                                                   3  => ( 0.003 ),
                                                   4  => ( 0.004 ),
                                                   5  => ( 0.005 ),
                                                   6  => ( 0.010 ),
                                                   7  => ( 0.025 ),
                                                   8  => ( 0.050 ),
                                                   9  => ( 0.075 ),
                                                   10 => ( 0.100 )),
                                    Optional  => ( 1  => ( 1.00 ),
                                                   2  => ( 0.05 ),
                                                   3  => ( 0.10 ),
                                                   4  => ( 0.25 ),
                                                   5  => ( 0.50 ),
                                                   6  => ( 0.75 ),
                                                   7  => ( 1.00 ),
                                                   8  => ( 2.00 ),
                                                   9  => ( 5.00 ),
                                                   10 => (10.00 ))) ;

    ------------------------------------------------------------
    --                                                        --
    --              TASK BODY Generic_Any_Time                --
    --                                                        --
    ------------------------------------------------------------
```

```
--
--    When not in use, the task is in the suspended state.
--
task body Generic_Any_Time is

    Start_Time   : Calendar.Time ;
    Stop_Time    : Calendar.Time ;
    My_Task_ID   : System.Task_ID ;
    My_Variables : Any_Time_Variables_Type ;

begin

    accept Initialize ( The_Variables : in  Any_Time_Variables_Type ;
                        The_Task_ID   : out System.Task_ID ;
                        Man_Duration  : out Duration ;
                        Opt_Duration  : out Duration )  do
        My_Task_ID   := V_XTasking.Current_Task ;
        The_Task_ID  := My_Task_ID ;
        My_Variables := The_Variables ;
        Man_Duration := Durations ( Mandatory, The_Variables.Kind ) ;
        Opt_Duration := Durations ( Optional , The_Variables.Kind ) ;
    end Initialize ;

    V_Xtasking.Suspend_Task ( My_Task_ID ) ;

    loop
        --
        -- Note that a 'while loop' is used and NOT a 'delay' statement.
        -- A delay statement would allow the operating system to suspend
        -- this task and it would not consume any CPU time.  For the
        -- purposes of simulating a 'real' task, it must consume CPU time.
        --
        select
            accept Change_Variables
                           ( The_Variables : in  Any_Time_Variables_Type ;
                             Man_Duration  : out Duration ;
                             Opt_Duration  : out Duration ) do
                My_Variables := The_Variables ;
                Man_Duration := Durations ( Mandatory, The_Variables.Kind ) ;
                Opt_Duration := Durations ( Optional , The_Variables.Kind ) ;
            end Change_Variables ;
        else
                --
                --    Start_Time and Stop_Time are used for debugging
                -- and display purposes only.
                --
            Start_Time := Calendar.Clock ;
            My_Variables := Any_Time_Variables_Buffer.Get_Item
                                                    ( My_Task_ID ) ;
            if My_Variables.Continue then
                --
                --    First execute the mandatory part
                --
                Run_Delay_Loop ( Durations ( Mandatory,
                                             My_Variables.Kind )) ;
                Stop_Time := Calendar.Clock ;
                if My_Variables.Display then
                    Register_Time ( Any_Time, Start_Time, Stop_Time,
                                    My_Task_ID, Mandatory ) ;
                end if ;
                --
                --    Notify the Task Manager that the mandatory
                -- part is complete.
                --
                Task_Manager.Task_Complete ( My_Task_ID ) ;

                --
                --    Now execute the optional part until told to stop.
                --
                loop
                        My_Variables := Any_Time_Variables_Buffer.Get_Item
                                                    ( My_Task_ID ) ;
                        if My_Variables.Continue then
                                Run_Delay_Loop ( Durations ( Optional,
                                                 My_Variables.Kind )) ;
                        --
                        --    Notify Task Manager completed a cycle
                        --
                        Task_Manager.Task_Complete ( My_Task_ID ) ;
                        else
                                --
                        -- Right now the only stopping condition is if
                        -- the Task Manager explicitly sets continue
                        -- to FALSE.  Other conditions should be
                        -- added based upon solution quality.
                        --
                        Stop_Time := Calendar.Clock ;
                        if My_Variables.Display then
                        Register_Time ( Any_Time, Start_Time, Stop_Time,
                                        My_Task_ID, Optional ) ;
                        end if ;
                        exit ;
                        end if ;
                    end loop ;
                end if ;
            end select ;
            --
            --    Always return the task shell to the suspended state
            -- when not being used.
            --
            V_XTasking.Suspend_Task ( My_Task_ID ) ;
        end loop ;
    end Generic_Any_Time ;

    ------------------------------------------------------
    --                                                  --
    --                PROCEDURE Create                  --
    --                                                  --
    ------------------------------------------------------
    procedure Create  ( The_Variables : in  Any_Time_Variables_Type ;
                        The_Task_Ptr  : out Any_Time_Task_Ptr ;
                        The_Task_ID   : out System.Task_ID ;
                        Man_Duration  : out Duration ;
                        Opt_Duration  : out Duration ) is
        Temp_Ptr : Any_Time_Task_Ptr ;

    begin
        Temp_Ptr := new Generic_Any_Time ;
        Temp_Ptr.Initialize ( The_Variables, The_Task_ID,
                              Man_Duration, Opt_Duration ) ;
        The_Task_Ptr := Temp_Ptr;
    end Create ;

    ------------------------------------------------------
    --                                                  --
    --             PROCEDURE Store_Variables            --
    --                                                  --
    ------------------------------------------------------
    procedure Store_Variables ( The_Variables : in Any_Time_Variables_Type ;
                                The_Task_ID   : in System.Task_ID ) is
    begin
        Any_Time_Variables_Buffer.Put_Item ( The_Item => The_Variables,
                                             The_Index => The_Task_ID ) ;
    end Store_Variables ;
    ------------------------------------------------------
    --                                                  --
    --             PROCEDURE Remove_Variables           --
    --                                                  --
    ------------------------------------------------------
    procedure Remove_Variables ( The_Task_ID : in System.Task_ID ) is
    begin
        Any_Time_Variables_Buffer.Remove_Item ( The_Index => The_Task_ID ) ;
    end Remove_Variables ;
end Any_Time_Tasks ;
```

```
--*****************************************************************************--
--*                                                                           *--
--*                      Periodic_Tasks Package Spec                          *--
--*                                                                           *--
--*****************************************************************************--
--*                                                                           *--
--*   TITLE: Periodic_Tasks                                                   *--
--*                                                                           *--
--*   DATE: 1 December 1992                                                   *--
--*                                                                           *--
--*   VERSION: 1.0                                                            *--
--*                                                                           *--
--*   AUTHOR: Michael A. Whelan                                               *--
--*                                                                           *--
--*   LANGUAGE: Verdix Ada Version 6.0                                        *--
--*                                                                           *--
--*****************************************************************************--
with System,
     Global_Data_Types;
package Periodic_Tasks is

    ------------------------------------------------------------
    --                                                        --
    --                  TYPE DECLARATIONS                     --
    --                                                        --
    ------------------------------------------------------------
    subtype Periodic_Types is Integer range 1..10 ;
    type Periodic_Variables_Type is
        record
            Continue : Boolean := TRUE ;
            Display  : Boolean := FALSE ;
            Kind     : Integer ;
            Mode     : Global_Data_Types.Mode_Type ;
        end record ;

    ------------------------------------------------------------
    --                                                        --
    --       TASK TYPE Generic_Periodic DECLARATION           --
    --                                                        --
    ------------------------------------------------------------
    task type Generic_Periodic is
        entry Initialize ( The_Variables : in  Periodic_Variables_Type ;
                           The_Task_ID   : out System.Task_ID    ;
                           Man_Duration  : out Duration          ;
                           Opt_Duration  : out Duration ) ;
        entry Change_Variables ( The_Variables : in Periodic_Variables_Type ;
                                 Man_Duration  : out Duration ;
                                 Opt_Duration  : out Duration ) ;
    end Generic_Periodic ;

    type Periodic_Task_Ptr is access Generic_Periodic ;

    procedure Create  ( The_Variables : in  Periodic_Variables_Type ;
                        The_Task_Ptr  : out Periodic_Task_Ptr ;
                        The_Task_ID   : out System.Task_ID    ;
                        Man_Duration  : out Duration          ;
                        Opt_Duration  : out Duration ) ;

    procedure Store_Variables ( The_Variables : in Periodic_Variables_Type ;
                                The_Task_ID   : in System.Task_ID ) ;

    procedure Remove_Variables ( The_Task_ID : in System.Task_ID ) ;

end Periodic_Tasks ;
```

```
--*****************************************************************************--
--*                                                                           *--
--*                     Periodic_Tasks Package Body                           *--
--*                                                                           *--
--*****************************************************************************--
--*                                                                           *--
--*   TITLE: Periodic_Tasks                                                   *--
--*                                                                           *--
--*   DATE: 1 December 1992                                                   *--
--*                                                                           *--
--*   VERSION: 1.0                                                            *--
--*                                                                           *--
--*   AUTHOR: Michael A. Whelan                                               *--
--*                                                                           *--
--*   LANGUAGE: Verdix Ada Version 6.0                                        *--
--*                                                                           *--
--*****************************************************************************--
with Calendar,
     Global_Data_Types,
     System,
     Task_Control_Buffer,
     Support_Functions,
     V_XTasking ;
use  Calendar,
     Global_Data_Types,
     System,
     Support_Functions,
     V_XTasking ;

package body Periodic_Tasks is

    ------------------------------------------------------------
    --                                                        --
    --                  TYPE DECLARATIONS                     --
    --                                                        --
    ------------------------------------------------------------
    type Durations_Type is array ( Global_Data_Types.Mode_Type,
                                   Periodic_Types ) of Duration ;

    package Periodic_Variables_Buffer is new Task_Control_Buffer
                                        ( Item  => Periodic_Variables_Type,
                                          Index => System.Task_ID ) ;

    ------------------------------------------------------------
    --                                                        --
    --                OBJECT DECLARATIONS                     --
    --                                                        --
    ------------------------------------------------------------
    Durations : Durations_Type := ( Mandatory => ( 1  => ( 0.0005 ),
                                                   2  => ( 0.0010 ),
                                                   3  => ( 0.0015 ),
                                                   4  => ( 0.0020 ),
                                                   5  => ( 0.0025 ),
                                                   6  => ( 0.0050 ),
                                                   7  => ( 0.0075 ),
                                                   8  => ( 0.0080 ),
                                                   9  => ( 0.0090 ),
                                                   10 => ( 0.0100 )),
                                    Optional  => ( 1  => ( 0.001 ),
                                                   2  => ( 0.002 ),
                                                   3  => ( 0.003 ),
                                                   4  => ( 0.004 ),
                                                   5  => ( 0.005 ),
                                                   6  => ( 0.010 ),
                                                   7  => ( 0.015 ),
                                                   8  => ( 0.020 ),
                                                   9  => ( 0.030 ),
                                                   10 => ( 0.040 ))) ;

    task body Generic_Periodic is

        Start_Time   : Calendar.Time ;
```

```
        Stop_Time    : Calendar.Time ;
        My_Task_ID   : System.Task_ID ;
        My_Variables : Periodic_Variables_Type ;

    begin
    --
    -- Get my TASK_ID number so I can request the parameters needed
    -- by this task to execute.  Note that this procedure call is
    -- specific to Verdix Ada.
    --
        accept Initialize ( The_Variables : in  Periodic_Variables_Type ;
                            The_Task_ID   : out System.Task_ID ;
                            Man_Duration  : out Duration ;
                            Opt_Duration  : out Duration )  do
            My_Task_ID  := V_Xtasking.Current_Task ;
            The_Task_ID  := My_Task_ID ;
            My_Variables := The_Variables ;
            Man_Duration := Durations ( Mandatory, The_Variables.Kind ) ;
            Opt_Duration := Durations ( Optional , The_Variables.Kind ) ;
        end Initialize ;

        V_Xtasking.Suspend_Task ( My_Task_ID ) ;

        loop
            --
            -- Check on object state would go here but now this loop
            -- simply wastes time as defined in the variables contained in
            -- the record structure Controls.  If the Controls.Mode variable
            -- is set to mandatory, then the time wasted is equal to the value
            -- of Mandatory_Duration.  If the Controls.Mode variable is set to
            -- optional, then the time wastes is equal to the value in Optional
            --  Duration.
            -- Note that a 'while loop' is used and NOT a 'delay' statement.
            -- A delay statement would allow the operating system to suspend
            -- this task and it would not consume any CPU time.  For the
            -- purposes of simulating a 'real' task, it must consume CPU time.
            --
            select
                accept Change_Variables ( The_Variables : in
Periodic_Variables_Type ;
                                          Man_Duration  : out Duration ;
                                          Opt_Duration  : out Duration ) do
                    My_Variables := The_Variables ;
                    Man_Duration := Durations ( Mandatory, The_Variables.Kind ) ;
                    Opt_Duration := Durations ( Optional , The_Variables.Kind ) ;
                end Change_Variables ;
            else
                Start_Time := Calendar.Clock ;
                My_Variables := Periodic_Variables_Buffer.Get_Item
                                                          ( My_Task_ID ) ;
                case My_Variables.Mode is
                    when Mandatory =>
                        Run_Delay_Loop ( Durations ( Mandatory, My_Variables.Kind
)) ;
                    when Optional =>
                        Run_Delay_Loop ( Durations ( Mandatory, My_Variables.Kind
)) ;
                        Run_Delay_Loop ( Durations ( Optional , My_Variables.Kind
)) ;
                end case ;
                Stop_Time := Calendar.Clock ;
                if My_Variables.Display then
                    Register_Time ( Periodic, Start_Time, Stop_Time,
                                    My_Task_ID, My_Variables.Mode ) ;
                end if ;
            end select ;
            V_XTasking.Suspend_Task ( My_Task_ID ) ;
        end loop ;
     end Generic_Periodic ;

    ------------------------------------------------------------
```

```
    --                                                        --
    --                    PROCEDURE Create                    --
    --                                                        --
    ------------------------------------------------------------
    procedure Create  ( The_Variables : in  Periodic_Variables_Type ;
                        The_Task_Ptr  : out Periodic_Task_Ptr ;
                        The_Task_ID   : out System.Task_ID ;
                        Man_Duration  : out Duration ;
                        Opt_Duration  : out Duration ) is

        Temp_Ptr : Periodic_Task_Ptr ;

    begin
        Temp_Ptr := new Generic_Periodic ;
        Temp_Ptr.Initialize ( The_Variables, The_Task_ID,
                              Man_Duration, Opt_Duration ) ;
        The_Task_Ptr := Temp_Ptr ;
    end Create ;

    ------------------------------------------------------------
    --                                                        --
    --             PROCEDURE Store_Variables                  --
    --                                                        --
    ------------------------------------------------------------
    procedure Store_Variables ( The_Variables : in Periodic_Variables_Type ;
                                The_Task_ID   : in System.Task_ID ) is
    begin
        Periodic_Variables_Buffer.Put_Item ( The_Item => The_Variables,
                                             The_Index => The_Task_ID ) ;
    end Store_Variables ;

    ------------------------------------------------------------
    --                                                        --
    --             PROCEDURE Remove_Variables                 --
    --                                                        --
    ------------------------------------------------------------
    procedure Remove_Variables ( The_Task_ID : in System.Task_ID ) is
    begin
        Periodic_Variables_Buffer.Remove_Item ( The_Index => The_Task_ID ) ;
    end Remove_Variables ;

end Periodic_Tasks ;
```

```
--*****************************************************************************--
--*                                                                           *--
--*                      Singular_Tasks Package Spec                          *--
--*                                                                           *--
--*****************************************************************************--
--*                                                                           *--
--*   TITLE: Singular_Tasks                                                   *--
--*                                                                           *--
--*   DATE: 1 December 1992                                                   *--
--*                                                                           *--
--*   VERSION: 1.0                                                            *--
--*                                                                           *--
--*   AUTHOR: Michael A. Whelan                                               *--
--*                                                                           *--
--*   LANGUAGE: Verdix Ada Version 6.0                                        *--
--*                                                                           *--
--*****************************************************************************--
with System,
     Global_Data_Types ;
use  System,
     Global_Data_Types ;
package Singular_Tasks is

    ------------------------------------------------------------
    --                                                        --
    --                  TYPE DECLARATIONS                     --
    --                                                        --
    ------------------------------------------------------------
    subtype Singular_Types is Integer range 1..10 ;
    type Singular_Variables_Type is
        record
            Display  : Boolean := FALSE ;
            Continue : Boolean := TRUE ;
            Mode     : Global_Data_Types.Mode_Type ;
            Kind     : Integer ;
        end record ;

    ------------------------------------------------------------
    --                                                        --
    --     TASK TYPE Generic_Singular DECLARATION             --
    --                                                        --
    ------------------------------------------------------------
   task type Generic_Singular is
        entry Initialize ( The_Variables : in  Singular_Variables_Type ;
                           The_Task_ID   : out System.Task_ID ;
                           Man_Duration  : out Duration ;
                           Opt_Duration  : out Duration ) ;
        entry Change_Variables ( The_Variables : in  Singular_Variables_Type ;
                                 Man_Duration  : out Duration ;
                                 Opt_Duration  : out Duration ) ;
    end Generic_Singular ;
    type Singular_Task_Ptr is access Generic_Singular ;

    procedure Create  ( The_Variables : in  Singular_Variables_Type ;
                        The_Task_Ptr  : out Singular_Task_Ptr ;
                        The_Task_ID   : out System.Task_ID ;
                        Man_Duration  : out Duration ;
                        Opt_Duration  : out Duration ) ;

    procedure Store_Variables ( The_Variables : in Singular_Variables_Type ;
                                The_Task_ID   : in System.Task_ID ) ;

    procedure Remove_Variables ( The_Task_ID : in System.Task_ID ) ;

end Singular_Tasks ;
```

```
--*****************************************************************************--
--*                                                                           *--
--*                    Singular_Tasks Package Body                            *--
--*                                                                           *--
--*****************************************************************************--
--*                                                                           *--
--*   TITLE: Singular_Tasks                                                   *--
--*                                                                           *--
--*   DATE: 1 December 1992                                                   *--
--*                                                                           *--
--*   VERSION: 1.0                                                            *--
--*                                                                           *--
--*   AUTHOR: Michael A. Whelan                                               *--
--*                                                                           *--
--*   LANGUAGE: Verdix Ada Version 6.0                                        *--
--*                                                                           *--
--*****************************************************************************--
with Calendar,
     Support_Functions,
     Task_Control_Buffer,
     Task_Manager_Package,
     V_XTasking ;
use  Calendar,
     Support_Functions,
     Task_Manager_Package,
     V_XTasking ;

package body Singular_Tasks is

    ------------------------------------------------------------
    --                                                        --
    --                  TYPE DECLARATIONS                     --
    --                                                        --
    ------------------------------------------------------------
    type Durations_Type is array ( Global_Data_Types.Mode_Type,
                                   Singular_Types ) of Duration ;

    package Singular_Variables_Buffer is new Task_Control_Buffer
                                     ( Item  => Singular_Variables_Type,
                                       Index => System.Task_ID ) ;

    Durations : Durations_Type := ( Mandatory => ( 1  => ( 0.0005 ),
                                                   2  => ( 0.0010 ),
                                                   3  => ( 0.0015 ),
                                                   4  => ( 0.0020 ),
                                                   5  => ( 0.0025 ),
                                                   6  => ( 0.0050 ),
                                                   7  => ( 0.0075 ),
                                                   8  => ( 0.0100 ),
                                                   9  => ( 0.0500 ),
                                                   10 => ( 0.1000 )),
                                    Optional  => ( 1  => ( 0.01 ),
                                                   2  => ( 0.05 ),
                                                   3  => ( 0.10 ),
                                                   4  => ( 0.25 ),
                                                   5  => ( 0.50 ),
                                                   6  => ( 0.75 ),
                                                   7  => ( 1.00 ),
                                                   8  => ( 2.00 ),
                                                   9  => ( 5.00 ),
                                                   10 => (10.00 ))) ;

    ------------------------------------------------------------
    --                                                        --
    --             TASK BODY Generic_Singular                 --
    --                                                        --
    ------------------------------------------------------------
    task body Generic_Singular is

        Start_Time   : Calendar.Time ;
        Stop_Time    : Calendar.Time ;
        My_Task_ID   : System.Task_ID ;
```

```
    My_Variables : Singular_Variables_Type ;

begin

    accept Initialize ( The_Variables : in  Singular_Variables_Type ;
                        The_Task_ID   : out System.Task_ID ;
                        Man_Duration  : out Duration ;
                        Opt_Duration  : out Duration )  do
        My_Task_ID   := V_Xtasking.Current_Task ;
        The_Task_ID  := My_Task_ID ;
        My_Variables := The_Variables ;
        Man_Duration := Durations ( Mandatory, The_Variables.Kind ) ;
        Opt_Duration := Durations ( Optional , The_Variables.Kind ) ;
    end Initialize ;

    V_Xtasking.Suspend_Task ( My_Task_ID ) ;

    loop
        --
        -- Note that a 'while loop' is used and NOT a 'delay' statement.
        -- A delay statement would allow the operating system to suspend
        -- this task and it would not consume any CPU time.  For the
        -- purposes of simulating a 'real' task, it must consume CPU time.
        --
        select
            accept Change_Variables
                            ( The_Variables : in Singular_Variables_Type ;
                              Man_Duration  : out Duration ;
                              Opt_Duration  : out Duration ) do
                My_Variables := The_Variables ;
                Man_Duration := Durations ( Mandatory, The_Variables.Kind ) ;
                Opt_Duration := Durations ( Optional , The_Variables.Kind ) ;
            end Change_Variables ;
        else
            Start_Time := Calendar.Clock ;
            My_Variables := Singular_Variables_Buffer.Get_Item
                                                    ( My_Task_ID ) ;
            if My_Variables.Continue then
                --
                --    Execute mandatory part
                --
                Run_Delay_Loop ( Durations ( Mandatory,
                                             My_Variables.Kind )) ;
                Stop_Time := Calendar.Clock ;
                if My_Variables.Display then
                    Register_Time ( Singular, Start_Time, Stop_Time,
                                    My_Task_ID, Mandatory ) ;
                end if ;
                --
                --    Notify Task Manager that mandatory part is
                -- complete.
                --
                Task_Manager.Task_Complete ( My_Task_ID ) ;
                --
                --    Check if  should execute optional part
                --
                My_Variables := Singular_Variables_Buffer.Get_Item
                                                    ( My_Task_ID ) ;
                if My_Variables.Continue then
                    --
                    --    Execute the optional part.
                    --
                    Run_Delay_Loop ( Durations ( Optional,
                                                 My_Variables.Kind )) ;
                    Stop_Time := Calendar.Clock ;
                    if My_Variables.Display then
                        Register_Time ( Singular, Start_Time, Stop_Time,
                                        My_Task_ID, Optional ) ;
                    end if ;
                    --
                    --   Notify Task Manager optional part is complete
                    --
                    Task_Manager.Task_Complete ( My_Task_ID ) ;
                end if ;
            end if ;
        end select ;
        --
        --  Since the dispatcher assumes that all tasks that it
        -- starts up have been suspended, we must leave this task in
        -- the suspended state.
        --
        V_XTasking.Suspend_Task ( My_Task_ID ) ;
    end loop ;
end Generic_Singular ;

-------------------------------------------------------------
--                                                         --
--                  PROCEDURE Create                       --
--                                                         --
-------------------------------------------------------------
procedure Create  ( The_Variables : in  Singular_Variables_Type ;
                    The_Task_Ptr  : out Singular_Task_Ptr ;
                    The_Task_ID   : out System.Task_ID ;
                    Man_Duration  : out Duration ;
                    Opt_Duration  : out Duration ) is
    Temp_Ptr : Singular_Task_Ptr ;

begin
    Temp_Ptr := new Generic_Singular ;
    Temp_Ptr.Initialize ( The_Variables, The_Task_ID,
                          Man_Duration, Opt_Duration ) ;
    The_Task_Ptr := Temp_Ptr ;
end Create ;

-------------------------------------------------------------
--                                                         --
--             PROCEDURE Store_Variables                   --
--                                                         --
-------------------------------------------------------------
procedure Store_Variables ( The_Variables : in Singular_Variables_Type ;
                            The_Task_ID   : in System.Task_ID ) is
begin
    Singular_Variables_Buffer.Put_Item ( The_Item => The_Variables,
                                         The_Index => The_Task_ID ) ;
end Store_Variables ;

-------------------------------------------------------------
--                                                         --
--             PROCEDURE Remove_Variables                  --
--                                                         --
-------------------------------------------------------------
procedure Remove_Variables ( The_Task_ID : in System.Task_ID ) is
begin
    Singular_Variables_Buffer.Remove_Item ( The_Index => The_Task_ID ) ;
end Remove_Variables ;

end Singular_Tasks ;
```

```
--*****************************************************************************--
--*                                                                           *--
--*                          Support_Functions                                *--
--*                                                                           *--
--*****************************************************************************--
--*                                                                           *--
--*   TITLE: Support_Functions                                                *--
--*                                                                           *--
--*   DATE: 1 December 1992                                                   *--
--*                                                                           *--
--*   VERSION: 1.0                                                            *--
--*                                                                           *--
--*   AUTHOR: Michael A. Whelan                                               *--
--*                                                                           *--
--*   LANGUAGE: Verdix Ada Version 6.0                                        *--
--*                                                                           *--
--*****************************************************************************--
--
--    These procedures support the Task Type packages.  The procedures allow
-- the Task Types to 1) consume CPU time for a specified amount of time, and
-- 2) print out the execution time of the task.
--
with System,
     Text_IO,
     Calendar,
     Global_Data_Types,
     Unchecked_Conversion ;
use calendar ;
package Support_Functions is

    procedure Run_Delay_Loop ( Delay_For : in Duration ) ;
    procedure Register_Time ( Task_Kind   : in Global_Data_Types.Task_Kind_Type ;
                              Start_Time  : in Calendar.Time ;
                              Stop_Time   : in Calendar.Time ;
                              Task_To_Reg : in System.Task_ID ;
                              Task_Mode   : in Global_Data_Types.Mode_Type ) ;
end Support_Functions ;

package body Support_Functions is

    package Time_IO is new Text_IO.Fixed_IO ( Duration ) ;
    package Task_ID_IO is new Text_IO.Integer_IO ( Integer ) ;

    function To_A_Integer is new
           Unchecked_Conversion ( System.Task_ID, Integer ) ;

    procedure Run_Delay_Loop ( Delay_For : in Duration ) is
        Start_Delay_Time : Calendar.Time ;
        Current_Time     : Calendar.Time ;
    begin
        Start_Delay_Time := Calendar.Clock ;
        Current_Time     := Calendar.Clock ;
        while ( Current_Time - Start_Delay_Time ) < Delay_For  loop
            Current_Time := Calendar.Clock ;
        end loop ;
    end Run_Delay_Loop ;

    procedure Register_Time ( Task_Kind   : in Global_Data_Types.Task_Kind_Type ;
                              Start_Time  : in Calendar.Time ;
                              Stop_Time   : in Calendar.Time ;
                              Task_To_Reg : in System.Task_ID ;
                              Task_Mode   : in Global_Data_Types.Mode_Type ) is
    begin
        Text_IO.Put ( " " & Global_Data_Types.Task_Kind_Type'Image ( Task_Kind )
                      & " , ") ;
        Task_ID_IO.Put ( To_a_Integer ( Task_To_Reg ) ) ;
        Text_IO.Put ( " , ") ;
        Time_IO.Put ( Calendar.Seconds ( Start_Time ), 6, 5 ) ;
        Text_IO.Put ( " , " ) ;
        Time_IO.Put ( Calendar.Seconds ( Stop_Time ), 6, 5 ) ;
        Text_IO.Put ( " , " ) ;
        Text_IO.Put ( " " & Global_Data_Types.Mode_Type'Image ( Task_Mode )
                                                          & " ") ;
        Text_IO.New_Line ;
    end Register_Time ;

end Support_Functions ;
```

```
--*****************************************************************************--
--*                                                                           *--
--*                  Task_Control_Buffer Package                              *--
--*                                                                           *--
--*****************************************************************************--
--*                                                                           *--
--*   TITLE: Task_Control_Buffer                                              *--
--*                                                                           *--
--*   DATE: 1 December 1992                                                   *--
--*                                                                           *--
--*   VERSION: 1.0                                                            *--
--*                                                                           *--
--*   AUTHOR: Michael A. Whelan                                               *--
--*                                                                           *--
--*   LANGUAGE: Verdix Ada Version 6.0                                        *--
--*                                                                           *--
--*****************************************************************************--
--
--
generic
    type Item is private ;
    type Index is private ;
package Task_Control_Buffer is

    function  Get_Item ( The_Index : in Index ) return Item ;
    procedure Put_Item ( The_Index : in Index ; The_Item : in Item ) ;
    procedure Remove_Item ( The_Index : in Index ) ;

end Task_Control_Buffer ;

with Monitor;
package body Task_Control_Buffer is

    ------------------------------------------------------------
    --                                                        --
    --                TASK DECLARATIONS                       --
    --                                                        --
    ------------------------------------------------------------
    task Buffer is
        entry Check_In ( The_Index : in Index ; The_Item : out Item ) ;
        entry Put_Item ( The_Index : in Index ; The_Item : in  Item ) ;
        entry Remove_Item ( The_Index : in Index ) ;
    end Buffer ;

    task body Buffer is
        ------------------------------------------------------------
        --                                                        --
        --                  TYPE DECLARATIONS                     --
        --                                                        --
        ------------------------------------------------------------
        type Buffer_List_Element_Type ;
        type Buffer_List_Type is access Buffer_List_Element_Type ;

        type Buffer_List_Element_Type is
            record
                The_Index : Index ;
                The_Item  : Item  ;
                Next      : Buffer_List_Type := null ;
            end record ;

        type Buffer_Type is
            record
                Guard      : Monitor.Kind ;
                The_Buffer : Buffer_List_Type := null ;
            end record ;

        ------------------------------------------------------------
        --                                                        --
        --                OBJECT DECLARATIONS                     --
        --                                                        --
        ------------------------------------------------------------
        Buffer            : Buffer_Type ;
        Current           : Buffer_List_Type  := null ;
        Temporary_Pointer : Buffer_List_Type  := null ;
        Free_List         : Buffer_List_Type  := null ;
        Temp_Item         : Item ;

        ------------------------------------------------------------
        --                                                        --
        --              FUNCTION New_Pointer                      --
        --                                                        --
        ------------------------------------------------------------
        function New_Pointer return Buffer_List_Type is
        begin
            if Free_List = null then
                return new Buffer_List_Element_Type ;
            else
                Temporary_Pointer := Free_List ;
                Free_List := Temporary_Pointer.Next ;
                Temporary_Pointer.Next := null ;
                return Temporary_Pointer ;
            end if ;
        end New_Pointer ;

        ------------------------------------------------------------
        --                                                        --
        --              PROCEDURE Add_To_Free                     --
        --                                                        --
        ------------------------------------------------------------
        procedure Add_To_Free ( Pointer : in Buffer_List_Type ) is
        begin
            Temporary_Pointer := Pointer  ;
            Temporary_Pointer.Next := Free_list ;
            Free_List := Temporary_Pointer ;
        end Add_To_Free ;

        ------------------------------------------------------------
        --                                                        --
        --                PROCEDURE Retreive                      --
        --                                                        --
        ------------------------------------------------------------
        function Retreive ( The_Index : in Index ) return Item is
        begin
            Monitor.Start_Reading ( Buffer.Guard ) ;
            Current := Buffer.The_Buffer ;
            while Current /= null loop
                if Current.The_Index = The_Index then
                   Temp_Item := Current.The_Item ;
                   Monitor.Stop_Reading ( Buffer.Guard ) ;
                   return Temp_Item ;
                end if ;
                Current := Current.Next ;
            end loop ;
            Monitor.Stop_Reading ( Buffer.Guard ) ;
        end Retreive ;

        ------------------------------------------------------------
        --                                                        --
        --                   PROCEDURE Store                      --
        --                                                        --
        ------------------------------------------------------------
        procedure Store  ( The_Index : in Index ; The_Item : in Item ) is
            Not_Stored : Boolean := True ;
        begin
            Monitor.Start_Writing ( Buffer.Guard ) ;
            if Buffer.The_Buffer = null then
                Buffer.The_Buffer := New_Pointer ;
                Buffer.The_Buffer.The_Item := The_Item ;
                Buffer.The_Buffer.The_Index := The_Index ;
            else
                Current := Buffer.The_Buffer ;
                while Not_Stored loop
                    if Current.The_Index = The_Index then
                        Current.The_Item := The_Item ;
```

```
                Not_Stored := FALSE ;
            elsif Current.Next = null then
                Current.Next := New_Pointer ;
                Current.Next.The_Item := The_Item ;
                Current.Next.The_Index := The_Index ;
                Not_Stored := FALSE ;
            else
                Current := Current.Next ;
            end if ;
        end loop ;
    end if ;
    Monitor.Stop_Writing ( Buffer.Guard ) ;
end Store ;

------------------------------------------------------
--                                                  --
--                PROCEDURE Remove                  --
--                                                  --
------------------------------------------------------
procedure Remove ( The_Index : in Index ) is
begin
    Monitor.Start_Writing ( Buffer.Guard ) ;
    Current := Buffer.The_Buffer ;
    Temporary_Pointer := null ;
    while Current /= null loop
        if Current.The_Index = The_Index then
            if Temporary_Pointer = null then
                Buffer.The_Buffer := null ;
                Add_To_Free ( Current ) ;
            else
                Temporary_Pointer.Next := Current.Next ;
                Add_To_Free ( Current ) ;
            end if ;
            Monitor.Stop_Writing ( Buffer.Guard ) ;
            return ;
        end if ;
        Temporary_Pointer := Current ;
        Current := Current.Next ;
    end loop ;
    Monitor.Stop_Writing ( Buffer.Guard ) ;
end Remove ;

begin
    loop
        select
            accept Check_In ( The_Index : in Index ; The_Item : out Item ) do
                The_Item := Retreive ( The_Index ) ;
            end Check_In ;
        or
            accept Put_Item ( The_Index : in Index ; The_Item : in Item ) do
                Store ( The_Index, The_Item ) ;
            end Put_Item ;
        or
            accept Remove_Item ( The_Index : in Index ) do
                Remove ( The_Index ) ;
            end Remove_Item ;
        end select ;
    end loop ;
end Buffer ;

------------------------------------------------------
--                                                  --
--              FUNCTION Get_Item                   --
--                                                  --
------------------------------------------------------
function Get_Item ( The_Index : in Index ) return Item is
    The_Item : Item ;
begin
    Buffer.Check_In (The_Index, The_Item) ;
    return The_Item ;
end Get_Item ;

------------------------------------------------------
--                                                  --
--              PROCEDURE Put_Item                  --
--                                                  --
------------------------------------------------------
procedure Put_Item ( The_Index : in Index ; The_Item : in Item ) is
begin
    Buffer.Put_Item ( The_Index, The_Item ) ;
end Put_Item ;

------------------------------------------------------
--                                                  --
--              PROCEDURE Remove_Item               --
--                                                  --
------------------------------------------------------
procedure Remove_Item ( The_Index : in Index ) is
begin
    Buffer.Remove_Item ( The_Index ) ;
end Remove_Item ;

end Task_Control_Buffer ;
```

```
--*****************************************************************************--
--*                                                                           *--
--*                          IO_Process_Package                               *--
--*                                                                           *--
--*****************************************************************************--
--*                                                                           *--
--*   TITLE: IO_Process_Package                                               *--
--*                                                                           *--
--*   DATE: 1 December 1992                                                   *--
--*                                                                           *--
--*   VERSION: 1.0                                                            *--
--*                                                                           *--
--*   AUTHOR: Michael A. Whelan                                               *--
--*                                                                           *--
--*   LANGUAGE: Verdix Ada Version 6.0                                        *--
--*                                                                           *--
--*****************************************************************************--
package IO_Process_Package is

    task IO_Process is
        Pragma Priority ( 99 ) ;
        entry Initialize ;
    end IO_Process ;

end IO_Process_Package ;

with Reasoning_Process_Package,
     Calendar,
     V_XTasking,
     System,
     Text_IO,
     Random_Numbers,
     Unchecked_Conversion ;
use  Random_Numbers ;

package body IO_Process_Package is

    task body IO_Process is
        Random_Value : Float := 0.0 ;
        Event_Number : Integer ;
    begin
        accept Initialize ;
        My_Task_ID := V_Xtasking.Current_Task ;
        loop
            Random_Value := random ;
            while Random_Value > 1.0 or
                  Random_Value < 0.1 loop
                Random_Value := random ;
            end loop ;
            Event_Number := Integer ( 10.0 * Random_Value ) ;
            Text_IO.Put_Line (" IO Event Number " & integer'image(event_Number)) ;
            Reasoning_Process_Package.Reasoning_Process.RP_Event_Message
                    ( Event_Number, Calendar.Clock ) ;
            Delay ( 2.0 ) ;
        end loop ;

    end IO_Process ;

end IO_Process_Package ;
```

```
--*****************************************************************************--
--*                                                                           *--
--*              Reasoning_Process_Package Package Spec                       *--
--*                                                                           *--
--*****************************************************************************--
--*                                                                           *--
--*   TITLE: Reasoning_Process_Package                                        *--
--*                                                                           *--
--*   DATE: 1 December 1992                                                   *--
--*                                                                           *--
--*   VERSION: 1.0                                                            *--
--*                                                                           *--
--*   AUTHOR: Michael A. Whelan                                               *--
--*                                                                           *--
--*   LANGUAGE: Verdix Ada Version 6.0                                        *--
--*                                                                           *--
--*****************************************************************************--
with Calendar,
     Global_Data_Types ;
use  Calendar,
     Global_Data_Types ;

package Reasoning_Process_Package is

    task Reasoning_Process is
        Pragma Priority ( 50 ) ;
        entry Initialize ;
        entry Assert_Fact ( The_Fact_String : in String ) ;
        entry Task_Completed  ( The_Task : in Integer ;
                                The_Time : in Calendar.Time ) ;
        entry Missed_Deadline ( The_Task : in Integer ;
                                The_Time : in Calendar.Time ) ;
        entry Infeasible_Task ( The_Task : in Integer ) ;
        entry RP_Event_Message ( Event_Number : in Integer ;
                                 Event_Time   : in Calendar.Time ) ;
        entry Task_Manager_Status
                        ( Status_Time            : in Calendar.Time ;
                          Periodic_Utilization   : in Float ;
                          Mandatory_Utilization  : in Float ;
                          Optional_Utilization   : in Float ;
                          Required_Utilization   : in Float ;
                          Periodic_Condition     : in Periodic_Condition_Type ;
                          Num_Periodic_Tasks     : in Integer ;
                          Num_Non_Periodic_Tasks : in Integer ) ;
    end Reasoning_Process ;

end Reasoning_Process_Package ;
```

```
--*****************************************************************************--
--*                                                                           *--
--*                        Reasoning_Process_Body                             *--
--*                                                                           *--
--*****************************************************************************--
--*                                                                           *--
--*   TITLE: Reasoning_Process_Body                                           *--
--*                                                                           *--
--*   DATE: 1 December 1992                                                   *--
--*                                                                           *--
--*   VERSION: 1.0                                                            *--
--*                                                                           *--
--*   AUTHOR: Michael A. Whelan                                               *--
--*                                                                           *--
--*   LANGUAGE: Verdix Ada Version 6.0                                        *--
--*                                                                           *--
--*****************************************************************************--
with Embedded_Clips ; use Embedded_Clips ;
with CLIPS_Globals  ; use CLIPS_Globals ;
with Utility ;        use Utility ;
with Fact_Manager ;   use Fact_Manager ;
with Unchecked_Conversion ;
with Text_IO ;

package body Reasoning_Process_Package is

    function Float_To_Clips_Real is new Unchecked_Conversion
        ( Source => Float ,
          Target => CLIPS_Real ) ;

    function Integer_To_Clips_Long is new Unchecked_Conversion
        ( Source => Integer ,
          Target => CLIPS_Long ) ;

    task body Reasoning_Process is

        ------------------------------------------------------
        --                                                  --
        --                OBJECT DECLARATIONS               --
        --                                                  --
        ------------------------------------------------------
        Clips_Inited : boolean := False ;
        The_Fact : CLIPS_Globals.Fact ;
        Rules_Fired : integer ;
        The_String  : String (1..256) ;
        Temp_Time   : Calendar.Time ;
        Task_ID     : Integer ;

        PT, NPT,
        The_Event,
        The_Length,
        Temp_Integer  : integer ;

        Temp_Float,
        PU, MU, RU, OU  : Float := 0.0 ;
        PC              : Periodic_Condition_Type ;

    begin

        accept Initialize do
            init_clips ;
            set_conserve_pp_memory ( off ) ;
            load_constructs ( "test_rp_rules.clp" ) ;
            reset_clips ;
          -- set_watch_item( "facts" , TRUE ) ;
        end Initialize ;

        loop
            select
                --
                -- Asserts a fact into CLIPS of the form :
                --
```

```
--  "(Event_Message <The_Event> <The_Time> )"
--
-- where <The_Event> is an integer value
--       <The_Time> is the number of seconds since midnight.
--
-- This is a simple "stubbed" procedure used for testing.
--
accept RP_Event_Message ( Event_Number : in Integer ;
                          Event_Time   : in Calendar.Time ) do
    The_Event := Event_Number ;
    Temp_Time := Event_Time   ;
end RP_Event_Message ;

Temp_Float := Float ( Calendar.Seconds ( Temp_Time ) ) ;

The_Fact := Utility.Get_Segment ( 3 ) ;

Utility.Set_Segment_Kind (The_Fact, 1, WORD_OBJECT ) ;
Utility.Set_Segment_String ( The_Fact, 1, "Event_Message" ) ;
Utility.Set_Segment_Kind ( The_Fact, 2, INTEGER_OBJECT) ;
Utility.Set_Segment_Integer
          ( The_Fact, 2, Integer_to_Clips_Long( The_Event )) ;
Utility.Set_Segment_Kind ( The_Fact, 3, REAL_OBJECT ) ;
Utility.Set_Segment_Float
          ( The_Fact, 3, Float_To_Clips_Real( Temp_Float )) ;

Fact_Manager.Add_Fact ( The_Fact ) ;
or
--
-- Asserts a fact into CLIPS of the form :
--
--  "(Task_Manager_Status <The_Time> <PU> <MU> <OU> <RU>
--                        <CONDITION> <#PT> <#NPT> )"
--
-- where <The_Time> is the time this status is for given in
--                  the number of seconds since midnight
--       <PU> is the current periodic utilization
--
accept Task_Manager_Status
        ( Status_Time           : in Calendar.Time ;
          Periodic_Utilization  : in Float ;
          Mandatory_Utilization : in Float ;
          Optional_Utilization  : in Float ;
          Required_Utilization  : in Float ;
          Periodic_Condition    : in Periodic_Condition_Type ;
          Num_Periodic_Tasks    : in Integer ;
          Num_Non_Periodic_Tasks : in Integer ) do
    Temp_Time := Status_Time ;
    PU := Periodic_Utilization ;
    MU := Mandatory_Utilization ;
    OU := Optional_Utilization ;
    RU := Required_Utilization ;
    PC := Periodic_Condition ;
    PT := Num_Periodic_Tasks ;
    NPT := Num_Non_Periodic_Tasks ,
end Task_Manager_Status ;

Temp_Float := Float ( Calendar.Seconds ( Temp_Time ) ) ;

The_Fact := Utility.Get_Segment ( 9 ) ;
Utility.Set_Segment_Kind (The_Fact, 1, WORD_OBJECT ) ;
Utility.Set_Segment_String
          ( The_Fact, 1, "Task_Manager_Status" ) ;
Utility.Set_Segment_Kind (The_Fact, 2, REAL_OBJECT ) ;
Utility.Set_Segment_Float
          ( The_Fact, 2, Float_to_Clips_Real ( Temp_Float )) ;
Utility.Set_Segment_Kind ( The_Fact, 3, REAL_OBJECT) ;
Utility.Set_Segment_Float
          ( The_Fact, 3, Float_to_Clips_Real(PU)) ;
Utility.Set_Segment_Kind ( The_Fact, 4, REAL_OBJECT) ;
Utility.Set_Segment_Float
          ( The_Fact, 4, Float_to_Clips_Real(MU)) ;
Utility.Set_Segment_Kind ( The_Fact, 5, REAL_OBJECT) ;
Utility.Set_Segment_Float
          ( The_Fact, 5, Float_to_Clips_Real(OU)) ;
Utility.Set_Segment_Kind ( The_Fact, 6, REAL_OBJECT) ;
Utility.Set_Segment_Float
          ( The_Fact, 6, Float_to_Clips_Real(RU)) ;
Utility.Set_Segment_Kind ( The_Fact, 7, WORD_OBJECT) ;
Utility.Set_Segment_String
       ( The_Fact, 7,
         Periodic_Condition_Type'Image(PC) ) ;
Utility.Set_Segment_Kind ( The_Fact, 8, INTEGER_OBJECT) ;
Utility.Set_Segment_Integer
          ( The_Fact, 8, Integer_to_Clips_Long(PT)) ;
Utility.Set_Segment_Kind ( The_Fact, 9, INTEGER_OBJECT) ;
Utility.Set_Segment_Integer
          ( The_Fact, 9, Integer_to_Clips_Long(NPT)) ;

Fact_Manager.Add_Fact ( The_Fact ) ;
or
accept Infeasible_Task ( The_Task : in Integer ) do
    Task_ID := The_Task ;
end Infeasible_Task ;

The_Fact := Utility.Get_Segment ( 2 ) ;
Utility.Set_Segment_Kind  (The_Fact, 1, WORD_OBJECT ) ;
Utility.Set_Segment_String ( The_Fact, 1, "Infeasible_Task" ) ;
Utility.Set_Segment_Kind  (The_Fact, 2, INTEGER_OBJECT ) ;
Utility.Set_Segment_Integer
    ( The_Fact, 2, Integer_To_Clips_Long ( Task_ID )) ;

Fact_Manager.Add_Fact ( The_Fact ) ;
or
accept Missed_Deadline ( The_Task : in Integer ;
                         The_Time : in Calendar.Time ) do
    Task_ID   := The_Task ;
    Temp_Time := The_Time ;
end Missed_Deadline ;

Temp_Float := Float ( Calendar.Seconds ( Temp_Time ) ) ;

The_Fact := Utility.Get_Segment ( 3 ) ;
Utility.Set_Segment_Kind  (The_Fact, 1, WORD_OBJECT ) ;
Utility.Set_Segment_String ( The_Fact, 1, "Missed_Deadline" ) ;
Utility.Set_Segment_Kind  (The_Fact, 2, INTEGER_OBJECT ) ;
Utility.Set_Segment_Integer
       ( The_Fact, 2, Integer_To_Clips_Long ( Task_ID )) ;
Utility.Set_Segment_Kind  (The_Fact, 3, REAL_OBJECT ) ;
Utility.Set_Segment_Float
          ( The_Fact, 3, Float_To_Clips_Real( Temp_Float )) ;

Fact_Manager.Add_Fact ( The_Fact ) ;
or
accept Task_Completed ( The_Task : in Integer ;
                        The_Time : in Calendar.Time ) do
    Task_ID   := The_Task ;
    Temp_Time := The_Time ;
end Task_Completed ;

Temp_Float := Float ( Calendar.Seconds ( Temp_Time ) ) ;

The_Fact := Utility.Get_Segment ( 3 ) ;
Utility.Set_Segment_Kind  (The_Fact, 1, WORD_OBJECT ) ;
Utility.Set_Segment_String ( The_Fact, 1, "Task_Completed" ) ;
Utility.Set_Segment_Kind  (The_Fact, 2, INTEGER_OBJECT ) ;
Utility.Set_Segment_Integer
          ( The_Fact, 2, Integer_To_Clips_Long ( Task_ID )) ;
Utility.Set_Segment_Kind  (The_Fact, 3, REAL_OBJECT ) ;
Utility.Set_Segment_Float
          ( The_Fact, 3, Float_To_Clips_Real( Temp_Float )) ;

Fact_Manager.Add_Fact ( The_Fact ) ;
or
```

```
            accept Assert_Fact ( The_Fact_String : in String ) do
                The_String (1..The_Fact_String'Length) := The_Fact_String ;
                The_Length := The_Fact_String'Length ;
            end Assert_Fact ;

            Embedded_CLIPS.Assert ( The_String( 1..The_Length ) ) ;
        or
            Delay 0.001 ;
            Rules_Fired := run_clips (1) ;

        end select ;

    end loop ;

end Reasoning_Process ;

end Reasoning_Process_Package ;
```



```
--*****************************************************************************--
--*                                                                           *--
--*                              User_Functions                               *--
--*                                                                           *--
--*****************************************************************************--
--*                                                                           *--
--*   TITLE: User_Functions                                                   *--
--*                                                                           *--
--*   DATE: 1 December 1992                                                   *--
--*                                                                           *--
--*   VERSION: 1.0                                                            *--
--*                                                                           *--
--*   AUTHOR: Michael A. Whelan                                               *--
--*                                                                           *--
--*   LANGUAGE: Verdix Ada Version 6.0                                        *--
--*                                                                           *--
--*****************************************************************************--
with Global_Data_Types;           use Global_Data_Types ;
with Task_Manager_Package ;       use Task_Manager_Package ;
with Calendar ;                   use Calendar ;
with Clips_Globals;               use Clips_Globals;
with Clips_Io;                    use Clips_Io;
with Evaluate;
with Fact_Manager ;               use Fact_Manager ;
with Memory_Manager;              use Memory_Manager;
with Symbol;
with Utility;                     use Utility;
with System ;                     use System ;
with Unchecked_Conversion ;

package body User_Functions is

    function Integer_To_CLIPS_Long is new Unchecked_Conversion
            ( Source => Integer ,
              Target => CLIPS_Long ) ;

    procedure Initialize_User_Functions is

    -- All user defined functions need to be specified here.
    -- The Define_Function procedure is from the Evaluate package
    -- Refer to page 2 in the CLIPS/Ada Advanced Programming Guide

    begin
        Evaluate.Define_Function
                        ( "add_task","add_task_op", "user_defined_op", 'l' ) ;
        Evaluate.Define_Function
                      ( "remove_task","remove_task_op", "user_defined_op", 'v' ) ;
        Evaluate.Define_Function
                    ( "modify_task","modify_task_op", "user_defined_op", 'v' ) ;
        Evaluate.Define_Function ( "new_periodic_utilization",
                        "new_periodic_utilization_op", "user_defined_op", 'v' ) ;
            -- first argument is a string representation of the name that
            -- will be used inside CLIPS/Ada rules.  The second argument is
            -- a string representation of the enumeration type defined by
            -- 'User_Defined_Op' type.  The third argument must be
            -- "user_defined_op".  The final arguement is the type of
            -- parameter returned by this sub-program.  Allowable values
            -- are 'l' for clips_long, 'd' for clips_real, 'c' for character,
            -- 's' for a pointer to a string, 'w' for a pointer to a character
            -- word, 'b' for a boolean, 'u' for a pointer to an unknown
            -- data type, 'm' for a pointer to a multifield variable, and
            -- 'v' for void.
    end Initialize_User_Functions ;

    procedure Evaluate_Function (
        The_Problem : In  Test ;
        The_Result  : out Values_Record ) is

        -- Add enumeration types here for each subprogram defined in
        -- Initialize User_Functions.
        type User_Defined_Op is ( add_task_op, remove_task_op, modify_task_op,
                                  new_periodic_utilization_op );
```

```
    The_User_Function : User_Defined_op ;
    Value             : Values_Record ;

    The_Type          : Task_Kind_Type ;
    Display_Flag      : Boolean ;
    The_Period        : Duration ;
    The_Deadline      : Calendar.Time ;
    The_Start_Time    : Calendar.Time ;
    The_TCB,
    The_Kind,
    The_Importance    : Integer ;

begin

    The_User_Function := User_Defined_Op'Value(
        The_Problem.Value.Function_Value.Defined_Name.all) ;

    -- Modify this case statement to call the appropriate subprogram.
    -- Each of the above defined enumerations types should have an
    -- entry in the case statement.

    case The_User_Function is

        when add_task_op =>
            --
            --  The CLIPS call should look like :
            --
            --  (add_task <arg1> <arg2> <arg3> <arg4> <arg5> <arg6> <arg7>)
            --
            --  Argument 1 -> Display Flag
            --          TRUE - Display the task's run times
            --          FALSE - Do not display it
            --  Argument 2 -> Task Type
            --          Can be either ANY_TIME, PERIODIC, SINGULAR
            --  Argument 3 -> Task Kind
            --          Can be any value
            --  Argument 4 -> Period
            --          Float value representing period
            --  Argument 5 -> Deadline
            --
            --  Argument 6 -> Start Time
            --
            --  Argument 7 -> Importance
            --
            The_Result.Kind := INTEGER_OBJECT ;
            --
            --    Get the first argument.  It should be a word that says
            -- whether or not to display the times of this task.  Thus
            -- it should be TRUE or FALSE.
            --
            Value := Get_Unknown_Argument ( The_Problem, 1 ) ;
            if Value.Kind = WORD_OBJECT and then
               Value.Symbol.Contents.all = "TRUE" then
                Display_Flag := TRUE ;
            else
                Display_Flag := FALSE ;
            end if ;
            --
            --   Get the second argument.  It should be a task type.
            -- Thus it should be either PERIODIC, ANY_TIME, or SINGULAR.
            --
            Value := Get_Unknown_Argument ( The_Problem, 2 ) ;
            if Value.Kind = WORD_OBJECT then
                if Value.Symbol.Contents.all = "ANY_TIME" then
                    The_Type := ANY_TIME ;
                elsif Value.Symbol.Contents.all = "PERIODIC" then
                    The_Type := PERIODIC ;
                elsif Value.Symbol.Contents.all = "SINGULAR" then
                    The_Type := SINGULAR ;
                else
                    raise Constraint_Error ;
                end if ;
            else
                raise Constraint_Error ;
            end if ;
            --
            --    Get the third argument.  It should be a value to indicate
            -- the task kind.  It should be an integer.
            --
            Value := Get_Unknown_Argument ( The_Problem, 3 ) ;
            if Value.Kind = INTEGER_OBJECT then
                The_Kind := Integer ( Value.Numeric_Int ) ;
            elsif Value.Kind = REAL_OBJECT then
                The_Kind := Integer ( Value.Numeric_Value ) ;
            else
                The_Kind := 1 ;
            end if ;
            --
            --    The fourth argument is supposed to be a float that
            -- represents the period of the task to add.
            --
            The_Period := Duration ( Get_Float_Argument ( The_Problem, 4 )) ;
            --
            --    The fifth argument is supposed to be a float that
            -- represents the deadline of this task.  The CLIPS_Ada function
            -- time returns the time now in seconds since midnight.  This
            -- is supposed to be the same thing, i.e. seconds to quit.
            --
            The_Deadline := Calendar.Time_Of
                                ( Year   ( Clock ),
                                  Month  ( Clock ),
                                  Day    ( Clock ),
                                  Duration
                                    ( Get_Float_Argument ( The_Problem, 5 )));
            The_Start_Time := Calendar.Time_Of
                                ( Year   ( Clock ),
                                  Month  ( Clock ),
                                  Day    ( Clock ),
                                  Duration
                                    ( Get_Float_Argument ( The_Problem, 6 )));
            --
            --
            --
            Value := Get_Unknown_Argument ( The_Problem, 7 ) ;
            if Value.Kind = INTEGER_OBJECT then
                The_Importance := Integer ( Value.Numeric_Int ) ;
            elsif Value.Kind = REAL_OBJECT then
                The_Importance := Integer ( Value.Numeric_Value ) ;
            else
                The_Importance := 1 ;
            end if ;
            Task_Manager.Add_Task ( Display_Flag, The_Type, The_Kind,
                                    The_Period, The_Deadline, The_Start_Time,
                                    The_Importance, The_TCB );
            The_Result.Numeric_Int :=  Integer_To_CLIPS_Long ( The_TCB ) ;

        when remove_task_op =>
            --
            --  (remove task 123)
            The_Result.Kind := VOID_OBJECT ;
            Value := Get_Unknown_Argument ( The_Problem, 1 ) ;
            if Value.Kind = INTEGER_OBJECT then
                The_TCB := Integer ( Value.Numeric_Int ) ;
                Task_Manager.Remove_Task ( The_TCB ) ;
            end if ;

        when modify_task_op =>
            --
            --  The CLIPS call should look like :
            --
            --    (modify_task <Arg1> <Arg2> <Arg3> <Arg4> <Arg5> <Arg6> )
            --
            --  Argument 1 -> New Display Flag
```

```
                --          TRUE - Display the task's run times
                --          FALSE - Do not display task's run times
                --  Argument 2 -> New Period
                --          Float value representing period
                --  Argument 3 -> New Deadline
                --
                --  Argument 4 -> New Start Time
                --
                --  Argument 5 -> Importance
                --
                --  Argument 6 -> The Task ID
                --
                The_Result.Kind := VOID_OBJECT ;
                Value := Get_Unknown_Argument ( The_Problem, 1 ) ;
                if Value.Kind = WORD_OBJECT and then
                   Value.Symbol.Contents.all = "TRUE" then
                    Display_Flag := TRUE ;
                else
                    Display_Flag := FALSE ;
                end if ;
                The_Period := Duration ( Get_Float_Argument ( The_Problem, 2 )) ;
                The_Deadline := Calendar.Time_Of
                                ( Year   ( Clock ),
                                  Month  ( Clock ),
                                  Day    ( Clock ),
                                  Duration
                                    ( Get_Float_Argument ( The_Problem, 3 )));
                The_Start_Time := Calendar.Time_Of
                                ( Year   ( Clock ),
                                  Month  ( Clock ),
                                  Day    ( Clock ),
                                  Duration
                                    ( Get_Float_Argument ( The_Problem, 4 )));
                Value := Get_Unknown_Argument ( The_Problem, 5 ) ;
                if Value.Kind = INTEGER_OBJECT then
                    The_Importance := Integer ( Value.Numeric_Int ) ;
                else
                    raise Constraint_Error ;
                end if ;
                Value := Get_Unknown_Argument ( The_Problem, 6 ) ;
                if Value.Kind = INTEGER_OBJECT then
                    The_TCB := Integer ( Value.Numeric_Int ) ;
                else
                    raise Constraint_Error ;
                end if ;
                Task_Manager.Modify_Task ( Display_Flag,   The_Period,
                                           The_Deadline,   The_Start_Time,
                                           The_Importance, The_TCB ) ;

            when new_periodic_utilization_op =>
                The_Result.Kind := VOID_OBJECT ;
                Task_Manager.New_Periodic_Utilization
                        ( Float ( Get_Float_Argument ( The_Problem, 1 ))) ;

        end case ;

    exception

        -- This exception handles the case of an improperly defined function
        when Constraint_Error =>
            Put ( Werror, The_Problem.Value.Function_Value.Defined_Name.all &
                  " is an invalid function." & CRLF ) ;
    end Evaluate_Function ;

end User_Functions ;
```

```
--*****************************************************************************--
--*                                                                           *--
--*                Task_Manager_Package Package Spec                          *--
--*                                                                           *--
--*****************************************************************************--
--*                                                                           *--
--*   TITLE: Task_Manager_Package                                             *--
--*                                                                           *--
--*   DATE: 1 December 1992                                                   *--
--*                                                                           *--
--*   VERSION: 1.0                                                            *--
--*                                                                           *--
--*   AUTHOR: Michael A. Whelan                                               *--
--*                                                                           *--
--*   LANGUAGE: Verdix Ada Version 6.0                                        *--
--*                                                                           *--
--*****************************************************************************--
with  System,
      Calendar,
      Global_Data_Types;
use   System,
      Calendar,
      Global_Data_Types;

package Task_Manager_Package is

    task Task_Manager is

        Pragma Priority ( 98 ) ;
        entry Initialize ;
        entry Print_Test_Results ( The_Test : in Integer ) ;
        entry Add_Task ( Display_Flag   : in Boolean ;
                         The_Type       : in Task_Kind_Type ;
                         The_Kind       : in Integer ;
                         The_Period     : in Duration ;
                         The_Deadline   : in Calendar.Time ;
                         The_Start_Time : in Calendar.Time ;
                         The_Importance : in Integer ;
                         New_Task_ID    : out Integer ) ;
        entry Modify_Task ( New_Display    : Boolean ;
                            New_Period     : Duration ;
                            New_Deadline   : Calendar.Time ;
                            New_Start_Time : Calendar.Time ;
                            New_Importance : Integer ;
                            The_Task_ID    : in Integer ) ;
        entry Remove_Task ( The_Task_ID    : in Integer ) ;
        entry New_Periodic_Utilization ( New_Util : in Float ) ;
        entry Task_Complete ( The_Task_ID : in System.Task_ID ) ;

    end Task_Manager ;

    Any_Time_Schedule_Error : exception ;
    Periodic_Schedule_Error : exception ;
    Singular_Schedule_Error : exception ;

end Task_Manager_Package ;
```

```
--*****************************************************************************--
--*                                                                           *--
--*                Task_Manager_Package Package Body                          *--
--*                                                                           *--
--*****************************************************************************--
--*                                                                           *--
--*   TITLE: Task_Manager_Package                                             *--
--*                                                                           *--
--*   DATE: 1 December 1992                                                   *--
--*                                                                           *--
--*   VERSION: 1.0                                                            *--
--*                                                                           *--
--*   AUTHOR: Michael A. Whelan                                               *--
--*                                                                           *--
--*   LANGUAGE: Verdix Ada Version 6.0                                        *--
--*                                                                           *--
--*****************************************************************************--
with Math,
     Text_IO,
     Any_Time_Tasks,
     Singular_Tasks,
     Periodic_Tasks,
     V_XTasking,
     Reasoning_Process_Package,
     Unchecked_Conversion,
     DeQue_Priority_Balking_Sequential_Unbounded_Managed_Iterator ;

use  Math,
     Text_IO,
     Any_Time_Tasks,
     Singular_Tasks,
     Periodic_Tasks,
     Reasoning_Process_Package,
     V_XTasking;

package body Task_Manager_Package is

       ------------------------------------------------------
       --                                                  --
---------                  NAMED NUMBERS                   ---------
       --                                                  --
       ------------------------------------------------------
    Twice_Task_Swap_Delay         : constant Float   := 0.0005 ;
    Task_Manager_Priority         : constant Integer := 98 ;
    Start_Periodic_Priority       : constant Integer := 90 ;
    Number_of_Periodic_Priorities : constant Integer := 80 ;
    Executing_Priority            : constant Integer := 9  ;
    Preempted_Priority            : constant Integer := 8  ;
    Discarded_Priority            : constant Integer := 1  ;
    Start_of_NP_Priorities        : constant Integer := 20 ;
    Bottom_Priority               : constant Integer := 7  ;

    Max_Periodic                  : constant Integer := 20 ;
    Max_Non_Periodic              : constant Integer := 10 ;

       ------------------------------------------------------
       --                                                  --
---------                TYPE DECLARATIONS                 ---------
       --                                                  --
       ------------------------------------------------------
    --
    --    These types are for recording min, max, and average times for
    -- testing purposes.
    --
    type Test_Values_Index_Type is ( MIN, MAX, SUM ) ;
    type Test_Values_Array is array (Test_Values_Index_Type,
                                     0..Max_Periodic,
                                     0..Max_Non_Periodic) of float;

    --
    -- Note that the use of the variant record requires that a default
    -- be given to the variant.  The program will not compile without
```

```
   -- this default because some unconstrained objects are declared.
   --
   type Status_Type is ( READY, EXECUTING, EXECUTING_MANDATORY,
                         EXECUTING_OPTIONAL, PREEMPTED_MANDATORY,
                         PREEMPTED_OPTIONAL, DISCARDED, COMPLETED ) ;

   type Task_Control_Block_Type ( Kind : Task_Kind_Type := PERIODIC ) ;
   type Task_Control_Block_Ptr is access Task_Control_Block_Type ;

   type Task_Control_Block_Type ( Kind : Task_Kind_Type := PERIODIC ) is
       record
           Deadline           : Calendar.Time ;
           Importance         : Integer             := 0 ;
           Latest_Start_Time  : Calendar.Time ;
           Mandatory_Duration : Duration            := 0.0 ;
           Next               : Task_Control_Block_Ptr ;
           Optional_Duration  : Duration            := 0.0 ;
           Period             : Duration            := 0.0 ;
           Priority           : System.Priority     := 0 ;
           Start_Time         : Calendar.Time ;
           Started_At         : Calendar.Time ;
           Status             : Status_Type         := COMPLETED ;
           Task_Kind          : Global_Data_Types.Task_Kind_Type ;
           Task_ID            : System.Task_ID ;
           Time_Remaining     : Duration            := 0.0 ;
           Integer_Task_ID    : Integer ;
           --
           -- A case is required for every kind of task that you wish
           -- to be able to create.
           --
           case Kind is
               when Global_Data_Types.Periodic =>
                   Periodic_Variables    : Periodic_Variables_Type ;
                   The_Periodic_Task_Ptr : Periodic_Task_Ptr ;
               when Global_Data_Types.Any_Time =>
                   Any_Time_Variables    : Any_Time_Variables_Type ;
                   The_Any_Time_Task_Ptr : Any_Time_Task_Ptr ;
               when Global_Data_Types.Singular =>
                   Singular_Variables    : Singular_Variables_Type ;
                   The_Singular_Task_Ptr : Singular_Task_Ptr ;
           end case ;
   end record ;

   type Task_List_Type is array ( Task_Kind_Type ) of Task_Control_Block_Ptr ;
   type Periodic_Bins_Type is array (1..Number_of_Periodic_Priorities)
                                                          of Integer ;

      ------------------------------------------------------------
      --                                                        --
---------          GENERIC INSTANTIATIONS                      ---------
      --                                                        --
      ------------------------------------------------------------
   package Int_IO is new Text_IO.Integer_IO ( Integer ) ;
   package Time_IO is new Text_IO.Fixed_IO ( Duration ) ;
   package Float_IO is new Text_IO.Float_IO ( Float ) ;

   function Task_ID_Of ( The_TCB : in Task_Control_Block_Ptr )
                                                    return Integer ;
   package Task_ID_Queue is
       new DeQue_Priority_Balking_Sequential_Unbounded_Managed_Iterator
                           ( Item        => Task_Control_Block_Ptr,
                             Priority    => Integer,
                             Priority_Of => Task_ID_Of,
                             "<"         => ">",
                             "<="        => ">=" ) ;

   function Starting_Time ( The_TCB : in Task_Control_Block_Ptr )
                                                    return Calendar.Time ;
   package Start_Time_Ordered_Queue is
       new DeQue_Priority_Balking_Sequential_Unbounded_Managed_Iterator
                           ( Item        => Task_Control_Block_Ptr,
                             Priority    => Calendar.Time,
                             Priority_Of => Starting_Time,
                             "<"         => Calendar.">",
                             "<="        => Calendar.">=" ) ;

   function Latest_Starting_Time ( The_TCB : in Task_Control_Block_Ptr )
                                                    return Calendar.Time ;
   package Latest_Start_Time_Ordered_Queue is
       new DeQue_Priority_Balking_Sequential_Unbounded_Managed_Iterator
                           ( Item        => Task_Control_Block_Ptr,
                             Priority    => Calendar.Time,
                             Priority_Of => Latest_Starting_Time,
                             "<"         => Calendar.">",
                             "<="        => Calendar.">=" ) ;

   function Deadline_Of ( The_TCB : in Task_Control_Block_Ptr )
                                                    return Calendar.Time ;
   package Deadline_Ordered_Queue is
       new DeQue_Priority_Balking_Sequential_Unbounded_Managed_Iterator
                           ( Item        => Task_Control_Block_Ptr,
                             Priority    => Calendar.Time,
                             Priority_Of => Deadline_Of,
                             "<"         => Calendar.">",
                             "<="        => Calendar.">=" ) ;

   function Importance_Of ( The_TCB : in Task_Control_Block_Ptr )
                                                    return Integer ;
   package Importance_Ordered_Queue is
       new DeQue_Priority_Balking_Sequential_Unbounded_Managed_Iterator
                           ( Item        => Task_Control_Block_Ptr,
                             Priority    => Integer,
                             Priority_Of => Importance_Of,
                             "<"         => ">",
                             "<="        => ">=" ) ;

   function Period_Of ( The_TCB : in Task_Control_Block_Ptr )
                                                    return Duration ;
   package Period_Ordered_Queue is
       new DeQue_Priority_Balking_Sequential_Unbounded_Managed_Iterator
                           ( Item        => Task_Control_Block_Ptr,
                             Priority    => Duration,
                             Priority_Of => Period_Of,
                             "<"         => ">",
                             "<="        => ">=" ) ;

      ------------------------------------------------------------
      --                                                        --
---------          OBJECT DECLARATIONS                         ---------
      --                                                        --
      ------------------------------------------------------------
   Test_Adds : Test_Values_Array := ( MIN => (OTHERS => (OTHERS => 1.0)),
                                      MAX => (OTHERS => (OTHERS => 0.0)),
                                      SUM =>  (OTHERS =>(OTHERS => 0.0)));
   Test_Rems : Test_Values_Array := ( MIN => (OTHERS => (OTHERS => 1.0)),
                                      MAX => (OTHERS => (OTHERS => 0.0)),
                                      SUM =>  (OTHERS =>(OTHERS => 0.0)));
   Test_Mods : Test_Values_Array := ( MIN => (OTHERS => (OTHERS => 1.0)),
                                      MAX => (OTHERS => (OTHERS => 0.0)),
                                      SUM =>  (OTHERS =>(OTHERS => 0.0)));
   Test_Comps : Test_Values_Array := ( MIN => (OTHERS => (OTHERS => 1.0)),
                                       MAX => (OTHERS => (OTHERS => 0.0)),
                                       SUM =>  (OTHERS =>(OTHERS => 0.0)));
   Test_PUs : Test_Values_Array := ( MIN => (OTHERS => (OTHERS => 1.0)),
                                     MAX => (OTHERS => (OTHERS => 0.0)),
                                     SUM =>  (OTHERS =>(OTHERS => 0.0)));

   Num_of_NP_Tasks,
   Number_of_Periodic_Tasks    : Integer := 0   ;

   Periodic_Utilization_Budget : Float   := 0.5 ;

   N,
   Additional_Utilization,
```

```
Periodic_Utilization,
Mandatory_Utilization,
Required_Utilization,
Optional_Utilization        : Float    := 0.0 ;

Bin_Number,
Current_Priority,
Periodic_Bin_Size,
Preempted_Value             : Integer  := 0 ;
Max_Period                  : Duration := 0.0 ;

Current_Time,
Min_Deadline,
Max_Deadline     : Calendar.Time ;
New_Mode         : Global_Data_Types.Mode_Type ;

Free_Task_List   : Task_List_Type ;

Temp_TCB,
New_TCB_Ptr,
Current_Task     : Task_Control_Block_Ptr ;

Periodic_Condition  : Periodic_Condition_Type := ALL_OPTIONAL ;

Tasks_By_ID                 : Task_ID_Queue.Deque ;
Tasks_By_ID_Queue_Back      : Task_ID_Queue.Location := Task_ID_Queue.BACK ;
Tasks_By_ID_Queue_Front     : Task_ID_Queue.Location := Task_ID_Queue.FRONT ;

Ready_Que          : Start_Time_Ordered_Queue.Deque ;
Ready_Queue_Back   : Start_Time_Ordered_Queue.Location
                           := Start_Time_Ordered_Queue.BACK ;
Ready_Queue_Front  : Start_Time_Ordered_Queue.Location
                           := Start_Time_Ordered_Queue.FRONT ;

Latest_Start_Time_Que         : Latest_Start_Time_Ordered_Queue.Deque ;
Latest_Start_Time_Queue_Back  : Latest_Start_Time_Ordered_Queue.Location
                              := Latest_Start_Time_Ordered_Queue.BACK ;
Latest_Start_Time_Queue_Front : Latest_Start_Time_Ordered_Queue.Location
                              := Latest_Start_Time_Ordered_Queue.FRONT ;


Deadline_Que,
Optional_Deadline_Que,
Mandatory_Deadline_Que      : Deadline_Ordered_Queue.Deque ;
Deadline_Queue_Back         : Deadline_Ordered_Queue.Location
                              := Deadline_Ordered_Queue.BACK ;
Deadline_Queue_Front        : Deadline_Ordered_Queue.Location
                              := Deadline_Ordered_Queue.FRONT ;

Temp_by_Period,
Tasks_by_Period    : Period_Ordered_Queue.Deque ;
Period_Queue_Back  : Period_Ordered_Queue.Location
                           := Period_Ordered_Queue.BACK ;
Period_Queue_Front : Period_Ordered_Queue.Location
                           := Period_Ordered_Queue.FRONT ;

Periodic_Importance_Que : Importance_Ordered_Queue.Deque ;
Importance_Queue_Back   : Importance_Ordered_Queue.Location
                           := Importance_Ordered_Queue.BACK ;
Importance_Queue_Front  : Importance_Ordered_Queue.Location
                           := Importance_Ordered_Queue.FRONT ;

Periodic_Bins,
Null_Periodic_Bins : Periodic_Bins_Type := ( others => 0 ) ;

function Task_ID_To_Integer is new Unchecked_Conversion
                        ( Source => System.Task_ID ,
                          Target => Integer ) ;

-------------------------------------------------------
--                                                   --
--               FUNCTION  Task_ID_Of                --
--                                                   --
```

```
-------------------------------------------------------
function Task_ID_Of( The_TCB : in Task_Control_Block_Ptr ) return Integer is
begin
    return The_TCB.Integer_Task_ID ;
end Task_ID_Of ;

-------------------------------------------------------
--                                                   --
--               FUNCTION  Starting_Time             --
--                                                   --
-------------------------------------------------------
function Starting_Time ( The_TCB : in Task_Control_Block_Ptr )
                                                  return Calendar.Time is
begin
    return The_TCB.Start_Time ;
end Starting_Time ;

-------------------------------------------------------
--                                                   --
--          FUNCTION  Lastest_Starting_Time          --
--                                                   --
-------------------------------------------------------
function Latest_Starting_Time ( The_TCB : in Task_Control_Block_Ptr )
                                                  return Calendar.Time is
begin
    return The_TCB.Latest_Start_Time ;
end Latest_Starting_Time ;

-------------------------------------------------------
--                                                   --
--                FUNCTION  Deadline_Of              --
--                                                   --
-------------------------------------------------------
function Deadline_Of ( The_TCB : in Task_Control_Block_Ptr )
                                                  return Calendar.Time is
begin
    return The_TCB.Deadline ;
end Deadline_Of ;

-------------------------------------------------------
--                                                   --
--               FUNCTION  Importance_Of             --
--                                                   --
-------------------------------------------------------
function Importance_Of ( The_TCB : in Task_Control_Block_Ptr )
                                                  return Integer is
begin
    return The_TCB.Importance ;
end Importance_Of ;

-------------------------------------------------------
--                                                   --
--                 FUNCTION  Period_Of               --
--                                                   --
-------------------------------------------------------
function Period_Of ( The_TCB : in Task_Control_Block_Ptr) return Duration is
begin
    return The_TCB.Period ;
end Period_Of ;

-------------------------------------------------------
--                                                   --
--     PROCEDURE  P_Tasks_Less_Than_P_Priorities     --
--                                                   --
-------------------------------------------------------
--
procedure P_Tasks_Less_Than_P_Priorities
                        ( New_Mode  : in Mode_Type ;
                          The_Queue : in Period_Ordered_Queue.Deque ) is

    Current_Priority : Integer := 0 ;
```

```
    Procedure Priorities_By_Period ( The_TCB  : in Task_Control_Block_Ptr ;
                                     Continue : out Boolean ) is
    begin
        if The_TCB.Periodic_Variables.Mode /= New_Mode then
            The_TCB.Periodic_Variables.Mode := New_Mode ;
            Periodic_Tasks.Store_Variables ( The_TCB.Periodic_Variables,
                                             The_TCB.Task_ID ) ;
        end if;
        The_TCB.Priority := Start_Periodic_Priority - Current_Priority ;
        Current_Priority := Current_Priority + 1 ;
        Continue := True ;
    end Priorities_By_Period ;
    procedure Assign_By_Period is new Period_Ordered_Queue.Iterate
                                     ( Process => Priorities_By_Period ) ;
begin
    Assign_By_Period ( The_Queue ) ;
end P_Tasks_Less_Than_P_Priorities ;

-------------------------------------------------------------
--                                                         --
--      PROCEDURE  P_Tasks_More_Than_P_Priorities          --
--                                                         --
-------------------------------------------------------------
--
procedure P_Tasks_More_Than_P_Priorities
                       ( New_Mode  : in Mode_Type ;
                         The_Queue : in Period_Ordered_Queue.Deque ) is

    Current_Priority : Integer := 0 ;
    Bin_Number       : Integer := 1 ;

    procedure Periodic_Priorites_by_Bin ( The_TCB : in Task_Control_Block_Ptr ;
                                          Continue : out Boolean ) is
    begin
        if The_TCB.Periodic_Variables.Mode /= New_Mode then
            The_TCB.Periodic_Variables.Mode := New_Mode ;
            Periodic_Tasks.Store_Variables ( The_TCB.Periodic_Variables,
                                             The_TCB.Task_ID ) ;
        end if;
         if Bin_Number < Number_of_Periodic_Priorities then
            if Periodic_Bins( Bin_Number ) < Periodic_Bin_Size then
                Periodic_Bins( Bin_Number ) := Periodic_Bins( Bin_Number )
                                               + 1 ;
            else
                Current_Priority := Current_Priority + 1 ;
                Bin_Number       := Bin_Number       + 1 ;
                Periodic_Bins( Bin_Number ) := 1 ;
            end if ;
        end if ;
        The_TCB.Priority := Start_Periodic_Priority - Current_Priority ;
        Continue := True ;
    end Periodic_Priorites_by_Bin ;

    procedure Traverse_By_Bins is new Period_Ordered_Queue.Iterate
                                  ( Process => Periodic_Priorites_by_Bin ) ;

begin
    Traverse_By_Bins ( The_Queue ) ;
end P_Tasks_More_Than_P_Priorities ;

-------------------------------------------------------------
--                                                         --
--      PROCEDURE  Print_Tasks_by_Period_Que               --
--                                                         --
-------------------------------------------------------------
--
procedure Print_The_Tasks_By_Period_Que
                              ( The_TCB  : in Task_Control_Block_Ptr ;
                                Continue : out Boolean ) is
    Task_ID : Integer ;
begin
    Int_IO.Put  ( The_TCB.Integer_Task_ID, 9 ) ;
    Time_IO.Put ( The_TCB.Period, 3, 5 ) ;
    Time_IO.Put ( The_TCB.Mandatory_Duration, 6, 4 ) ;
    Time_IO.Put ( The_TCB.Optional_Duration, 5, 4 ) ;
    Int_IO.Put  ( The_TCB.Periodic_Variables.Kind, 6 ) ;
    Int_IO.Put  ( The_TCB.Importance, 9) ;
    Int_IO.Put  ( Integer(The_TCB.Priority), 10 ) ;
    Text_IO.Put ( "    " &
                  Mode_Type'Image(The_TCB.Periodic_Variables.Mode) ) ;
    Text_IO.New_Line ;
    Continue := True ;
end Print_The_Tasks_By_Period_Que ;
procedure Print_Tasks_By_Period is new Period_Ordered_Queue.Iterate
                              ( Process => Print_The_Tasks_By_Period_Que ) ;

-------------------------------------------------------------
--                                                         --
--      PROCEDURE  Print_Non_Periodic_Tasks                --
--                                                         --
-------------------------------------------------------------
--
procedure Print_Only_Non_Periodic_Tasks
                                ( The_TCB  : in Task_Control_Block_Ptr ;
                                  Continue : out Boolean ) is
    Task_ID : Integer ;
begin
    if The_TCB.Task_Kind /= PERIODIC and then
       The_TCB.Status /= COMPLETED then
        Int_IO.Put  ( The_TCB.Integer_Task_ID, 9 ) ;
        Time_IO.Put ( The_TCB.Mandatory_Duration, 5, 3 ) ;
        Time_IO.Put ( The_TCB.Optional_Duration, 6, 3 ) ;
        Time_IO.Put ( The_TCB.Time_Remaining, 5, 3 ) ;
        Time_IO.Put ( Calendar.Seconds(The_TCB.Start_Time), 8, 3 ) ;
        Time_IO.Put ( Calendar.Seconds(The_TCB.Deadline), 7, 3 ) ;
        if The_TCB.Status = EXECUTING_MANDATORY or
           The_TCB.Status = EXECUTING_OPTIONAL then
            Time_IO.Put ( Calendar.Seconds(The_TCB.Started_At), 7, 3 ) ;
        else
            Text_IO.Put ( "    N/A    " ) ;
        end if ;
        Int_IO.Put  ( The_TCB.Importance, 8) ;
        Int_IO.Put  ( Status_Type'Pos(The_TCB.Status), 9) ;
        if The_TCB.Status = PREEMPTED_MANDATORY or
           The_TCB.Status = PREEMPTED_OPTIONAL then
            Time_IO.Put ( Calendar.Seconds(The_TCB.Latest_Start_Time),7,3 );
        end if ;
        Text_IO.New_Line ;
    end if;
    Continue := True ;
end Print_Only_Non_Periodic_Tasks ;
procedure Print_Non_Periodic_Tasks is new Task_ID_Queue.Iterate
                              ( Process => Print_Only_Non_Periodic_Tasks ) ;

   -------------------------------------------------------------
   --                                                         --
---------         INTERNAL PROCEDURES AND FUNCTIONS           ---------
   --                                                         --
   -------------------------------------------------------------
function Get_TCB ( Display_Flag   : in Boolean ;
                   The_Type       : in Task_Kind_Type ;
                   The_Kind       : in Integer ;
                   The_Period     : in Duration ;
                   The_Deadline   : in Calendar.Time ;
                   The_Start_Time : in Calendar.Time ;
                   The_Importance : in Integer )
                              return Task_Control_Block_Ptr is separate ;
function Find_TCB ( The_Task_ID : in Integer )
                              return Task_Control_Block_Ptr is separate ;
procedure Periodic_Priorities_By_Importance
           ( The_Queue  : in Importance_Ordered_Queue.Deque ;
             Temp_Queue : in out Period_Ordered_Queue.Deque ) is separate ;
procedure Some_Periodics_Optional
           ( The_Queue : in Importance_Ordered_Queue.Deque ) is separate ;
```

```
    procedure Assign_Periodic_Priorities is separate ;
    procedure Schedule ( The_TCB : in out Task_Control_Block_Ptr ) is separate ;
    procedure Un_Schedule ( The_TCB : in Task_Control_Block_Ptr ) is separate ;
    procedure Non_Periodic_Completed (The_TCB : in Task_Control_Block_Ptr)
                                                          is separate ;
    procedure Modify ( The_TCB        : in Task_Control_Block_Ptr;
                       New_Period     : Duration ;
                       New_Display    : Boolean ;
                       New_Deadline   : Calendar.Time ;
                       New_Start_Time : Calendar.Time ;
                       New_Importance : Integer ) is separate ;
    function Feasible ( The_TCB : in Task_Control_Block_Ptr )
                                          return boolean is separate ;
    function Modified_Feasible ( The_TCB        : in Task_Control_Block_Ptr;
                                 New_Deadline   : Calendar.Time ;
                                 New_Start_Time : Calendar.Time )
                                          return boolean is separate ;
    function Dispatch_Tasks return duration is separate ;
    procedure Print_Periodic_Tasks is separate ;
    procedure Print_Non_Periodic_Tasks is separate ;
    procedure Record_Times ( Start_Time : in Calendar.Time ;
                             Stop_Time  : in Calendar.Time ;
                             Num_NP_Ts  : in integer ;
                             Num_P_Ts   : in integer ;
                             The_Array  : in out Test_Values_Array ) is separate;
    procedure Print_Test_Times ( Test_Number : in integer ) is separate ;
    task body Task_Manager is separate ;

end Task_Manager_Package ;
```

```
separate ( Task_Manager_Package )

    ------------------------------------------------------------
    --                                                        --
    --              TASK BODY Task_Manager                    --
    --                                                        --
    ------------------------------------------------------------

task body Task_Manager is

    New_Task_ID,
    Temp_Num_NP_Tasks,
    The_New_Importance : Integer ;
    The_New_Display    : Boolean ;
    Add_Time,
    The_New_Period     : Duration ;
    Test_Stop_Time,
    Test_Start_Time,
    The_New_Deadline   : Calendar.Time ;
    The_New_Start_Time : Calendar.Time ;
    Completed_Task     : System.Task_ID ;

begin

    accept Initialize do

        Reasoning_Process.Task_Manager_Status ( Calendar.Clock,
                                Periodic_Utilization,
                                Mandatory_Utilization,
                                Optional_Utilization,
                                Required_Utilization,
                                Periodic_Condition,
                                Number_of_Periodic_Tasks,
                                Num_of_NP_Tasks ) ;
    end Initialize ;

    loop
        --   The order is explicitly defined with the use of the when guards
        -- so that add_task is done first, then modify_task, then remove_task,
        -- then the others.  You do not want to modify a task that has not
        -- yet been added and you do not want to remove a task that does not
        -- exist.
        --
        select

            accept Print_Test_Results ( The_Test : in Integer ) do
                if The_Test <= 6 then
                    Print_Test_Times (The_Test) ;
                elsif The_Test = 7 then
                    Print_Periodic_Tasks ;
                elsif The_Test = 8 then
                    Print_Non_Periodic_Tasks ;
                end if ;
            end Print_Test_Results ;

        or
            accept Task_Complete ( The_Task_ID : in System.Task_ID ) do
                Test_Start_Time := Calendar.Clock ;
                Completed_Task := The_Task_ID ;
            end Task_Complete ;

            Temp_TCB := Find_TCB ( Task_ID_To_Integer (Completed_Task) ) ;
            Non_Periodic_Completed ( Temp_TCB ) ;

            Test_Stop_Time := Calendar.Clock ;
            Record_Times ( Test_Start_Time, Test_Stop_Time, Num_of_NP_Tasks,
                           Number_Of_Periodic_Tasks, Test_Comps ) ;
            --
            --    The rendezvous here is only as long as it takes to create
            -- a TCB and return its pointer to the calling task.
            --
        or
```

```
when Task_Complete'Count = 0 =>
accept Add_Task ( Display_Flag   : in Boolean ;
                  The_Type       : in Task_Kind_Type ;
                  The_Kind       : in Integer ;
                  The_Period     : in Duration ;
                  The_Deadline   : in Calendar.Time ;
                  The_Start_Time : in Calendar.Time ;
                  The_Importance : in Integer ;
                  New_Task_ID    : out Integer ) do

    Test_Start_Time := Calendar.Clock ;
    Temp_TCB := Get_TCB ( Display_Flag, The_Type, The_Kind,
                          The_Period, The_Deadline, The_Start_Time,
                          The_Importance ) ;
    New_Task_ID := Temp_TCB.Integer_Task_ID ;

end Add_Task ;

--
--   First check and see if the new task is feasible.  If it is
-- then just schedule it and we're done.  If it is not, then it
-- still needs to be scheduled, but the reasoning process needs
-- to made aware of the problem.  Exactly what to do with a new
-- task that makes the task sets infeasible should be examined
-- more closely.
--

If Feasible ( Temp_TCB ) then
   Schedule ( Temp_TCB ) ;
else
   case Temp_TCB.Task_Kind is
       --
       --  Infeasible periodic tasks are still scheduled because
       -- at a future time they may become feasible.
       --
       when PERIODIC =>
           Text_IO.Put_Line ( ">>> Periodic Schedule Error <<<") ;
           Schedule ( Temp_TCB ) ;
       --
       -- Non-Periodic tasks are removed from the system.
       --
       when ANY_TIME =>
           Text_IO.Put_Line ( ">>> Any Time Schedule Error <<<") ;
           Un_Schedule ( Temp_TCB ) ;
           Reasoning_Process.Infeasible_Task
                       ( Temp_TCB.Integer_Task_ID ) ;

       when SINGULAR =>
           Text_IO.Put_Line ( ">>> Singular Schedule Error <<<") ;
           Un_Schedule ( Temp_TCB ) ;
           Reasoning_Process.Infeasible_Task
                       ( Temp_TCB.Integer_Task_ID ) ;
   end case ;

end if ;

Reasoning_Process.Task_Manager_Status ( Calendar.Clock,
                                        Periodic_Utilization,
                                        Mandatory_Utilization,
                                        Optional_Utilization,
                                        Required_Utilization,
                                        Periodic_Condition,
                                        Number_of_Periodic_Tasks,
                                        Num_of_NP_Tasks ) ;
Test_Stop_Time := Calendar.Clock ;
Record_Times ( Test_Start_Time, Test_Stop_Time, Num_of_NP_Tasks,
               Number_Of_Periodic_Tasks, Test_Adds ) ;

or
    when Task_Complete'Count = 0 and then
         Add_Task'Count = 0 =>
    accept Modify_Task ( New_Display    : Boolean ;
                         New_Period     : Duration ;
                         New_Deadline   : Calendar.Time ;
                         New_Start_Time : Calendar.Time ;
                         New_Importance : Integer ;
                         The_Task_ID    : in Integer ) do

        Test_Start_Time := Calendar.Clock ;
        Temp_TCB := Find_TCB ( The_Task_ID ) ;
        The_New_Period     := New_Period ;
        The_New_Display    := New_Display ;
        The_New_Deadline   := New_Deadline ;
        The_New_Start_Time := New_Start_Time ;
        The_New_Importance := New_Importance ;

    end Modify_Task ;

    If Modified_Feasible ( Temp_TCB, The_New_Deadline,
                           The_New_Start_Time ) then
        Modify   ( Temp_TCB, The_New_Period, The_New_Display,
                   The_New_Deadline, The_New_Start_Time,
                   The_New_Importance ) ;
    else
        Text_IO.Put_Line ( ">>> Modify Infeasible <<<") ;
        Reasoning_Process.Infeasible_Task ( Temp_TCB.Integer_Task_ID ) ;
    end if ;

    Reasoning_Process.Task_Manager_Status ( Calendar.Clock,
                                            Periodic_Utilization,
                                            Mandatory_Utilization,
                                            Optional_Utilization,
                                            Required_Utilization,
                                            Periodic_Condition,
                                            Number_of_Periodic_Tasks,
                                            Num_of_NP_Tasks ) ;
    Test_Stop_Time := Calendar.Clock ;
    Record_Times ( Test_Start_Time, Test_Stop_Time, Num_of_NP_Tasks,
                   Number_Of_Periodic_Tasks, Test_Mods ) ;
or
    when Task_Complete'Count = 0 and then
         Add_Task'Count = 0      and then
         Modify_Task'Count = 0 =>
    accept Remove_Task ( The_Task_ID : in Integer ) do

        Test_Start_Time := Calendar.Clock ;
        Temp_TCB :=  Find_TCB ( The_Task_ID ) ;

    end Remove_Task ;

    Un_Schedule ( Temp_TCB ) ;
    Reasoning_Process.Task_Manager_Status ( Calendar.Clock,
                                            Periodic_Utilization,
                                            Mandatory_Utilization,
                                            Optional_Utilization,
                                            Required_Utilization,
                                            Periodic_Condition,
                                            Number_of_Periodic_Tasks,
                                            Num_of_NP_Tasks ) ;
    Test_Stop_Time := Calendar.Clock ;
    if Temp_TCB.Task_Kind = PERIODIC then
        Record_Times ( Test_Start_Time, Test_Stop_Time, Num_of_NP_Tasks,
                       Number_Of_Periodic_Tasks + 1, Test_Rems ) ;
    else
        Record_Times ( Test_Start_Time, Test_Stop_Time,
                       Num_of_NP_Tasks + 1,
                       Number_Of_Periodic_Tasks, Test_Rems ) ;
    end if ;

or
    when Task_Complete'Count = 0 and then
         Add_Task'Count = 0      and then
         Modify_Task'Count = 0   and then
         Remove_Task'Count = 0 =>
```

```
        accept New_Periodic_Utilization ( New_Util : in Float ) do
            Test_Start_Time := Calendar.Clock ;
            Periodic_Utilization_Budget := New_Util ;
        end New_Periodic_Utilization ;

        if Number_of_Periodic_Tasks > 0 then
            N := float (Number_of_Periodic_Tasks) ;
            Required_Utilization := N * ( 2.0 ** (1.0 / N) -1.0 ) *
                                    Periodic_Utilization_Budget ;
            Assign_Periodic_Priorities ;
        end if ;
        Reasoning_Process.Task_Manager_Status ( Calendar.Clock,
                                                Periodic_Utilization,
                                                Mandatory_Utilization,
                                                Optional_Utilization,
                                                Required_Utilization,
                                                Periodic_Condition,
                                                Number_of_Periodic_Tasks,
                                                Num_of_NP_Tasks ) ;

        Test_Stop_Time := Calendar.Clock ;
        Record_Times ( Test_Start_Time, Test_Stop_Time, Num_of_NP_Tasks,
                       Number_Of_Periodic_Tasks, Test_PUs ) ;
    or
        --
        --   This delay statement first starts up any task that needs to be
        -- started and then delays until the next scheduling event needs
        -- to occur.  The structure of the Select statement allows calls
        -- to be made to the task manager while the delay is running.
        --
        Delay ( Dispatch_Tasks ) ;

    end select ;

  end loop ;

end Task_Manager ;
```



```
separate ( Task_Manager_Package )
  ------------------------------------------------------------
  --                                                        --
  --         PROCEDURE  Assign_Periodic_Priorities          --
  --                                                        --
  ------------------------------------------------------------
procedure Assign_Periodic_Priorities is

begin
  --
  --    Zero out the periodic bins array.
  --
  Periodic_Bins := Null_Periodic_Bins ;
  --
  --    The following if-then-elsif-then construct determines where
  -- along the "utilization" line the current required utilization
  -- falls.  If the Required_Utilization is below the mandatory utilization
  -- then only the most important tasks will be able to execute their
  -- mandatory parts.  If the required utilization is less than the
  -- optional utilization of the periodic task set, but greater than the
  -- mandatory utilization, then all periodic tasks will execute their
  -- mandatory parts and only the most important tasks will execute their
  -- optional parts.  If the required utilization is greater than the optional
  -- utilization of the current task set, then each task will execute both
  -- its mandatory part and its optional part.
  --
  --          Some             Some             All
  --    <-- Mandatory -->|<-- Optional -->|<-- Optional -->
  --
  --    0                                                 1
  --    |_________________|_________________|_________________|
  --                      |                 |
  --                Mandatory          Optional
  --               Utilization        Utilization
  --
  if Required_Utilization < Mandatory_Utilization then
      --
      --    If this condition is satisfied then we can only do some of the
      -- mandatory parts of some tasks and the others must be assigned
      -- to the bottom priority.
      --
      New_Mode := Global_Data_Types.Mandatory ;
      Periodic_Utilization := 0.0 ;
      Period_Ordered_Queue.Clear ( Temp_by_Period ) ;
      Periodic_Condition := SOME_MANDATORY ;
      Periodic_Priorities_By_Importance ( Periodic_Importance_Que,
                                          Temp_by_Period ) ;
      if Period_Ordered_Queue.Length_Of ( Temp_by_Period ) <=
                                          Number_of_Periodic_Priorities then
          P_Tasks_Less_Than_P_Priorities ( New_Mode, Temp_by_Period ) ;
      else
          P_Tasks_More_Than_P_Priorities ( New_Mode, Temp_by_Period ) ;
      end if ;

  elsif Required_Utilization < Optional_Utilization then
      --
      --    If we make it here, then all the tasks can execute their
      -- mandatory parts and some can execute their optional parts and the
      -- Periodic_Utilization is at least the Mandatory_Utilization.  So,
      -- first the modes of all tasks are set to mandatory and the priorities
      -- are assigned.  Next, the modes of the important tasks are set to
      -- optional util can't do anymore.
      --
      New_Mode := Global_Data_Types.Mandatory ;
      Periodic_Utilization := Mandatory_Utilization ;
      Periodic_Condition := SOME_OPTIONAL ;
      if Number_of_Periodic_Tasks <= Number_of_Periodic_Priorities then
          P_Tasks_Less_Than_P_Priorities ( New_Mode, Tasks_by_Period\ ) ;
      else
          P_Tasks_More_Than_P_Priorities ( New_Mode, Tasks_by_Period ) ;
      end if ;
      Some_Periodics_Optional ( Periodic_Importance_Que ) ;
```

```
    else
        --
        --    If we make it to this point then there is no problem with
        -- processor utilization and we can schedule all the tasks to run
        -- using both their mandatory and optional parts.
        --
        New_Mode := Global_Data_Types.OPTIONAL ;
        Periodic_Utilization := Optional_Utilization ;
        Periodic_Condition := ALL_OPTIONAL ;
        if Number_of_Periodic_Tasks <= Number_of_Periodic_Priorities then
           P_Tasks_Less_Than_P_Priorities ( New_Mode, Tasks_By_Period ) ;
        else
           P_Tasks_More_Than_P_Priorities ( New_Mode, Tasks_By_Period ) ;
        end if ;
    end if ;

--    case Periodic_Condition is
--       when SOME_MANDATORY =>
--          Text_IO.Put_Line (">>> Some_Mandatory <<<") ;
--       when SOME_OPTIONAL =>
--          Text_IO.Put_Line (">>> Some_Optional <<<") ;
--       when ALL_OPTIONAL =>
--          Text_IO.Put_Line (">>> All_Optional <<<") ;
--    end case ;

end Assign_Periodic_Priorities ;
```

```
separate ( Task_Manager_Package )
    ------------------------------------------------------------
    --                                                        --
    --             FUNCTION  Dispatch_Tasks                   --
    --                                                        --
    ------------------------------------------------------------

--
--   The job here is to first, kill any jobs that have exceeded their
-- deadlines and to start any other jobs that need to be started.  The
-- value return here is the lesser of the next deadline or the next
-- start time.
--
function Dispatch_Tasks return duration is

    The_TCB : Task_Control_Block_Ptr ;

    ------------------------------------------------------------
    --                                                        --
    --          PROCEDURE   Preempt_Current_Task              --
    --                                                        --
    ------------------------------------------------------------
    --
    --   To preempt the current task you need to change its priority and
    -- status and add it to the queues, Latest_start_time and Deadline.
    --
    procedure Preempt_Current_Task
                          ( New_Current_Task : in Task_Control_Block_Ptr ) is
    begin
        if Current_Task.Status = EXECUTING_OPTIONAL then
            Current_Task.Status := PREEMPTED_OPTIONAL ;
        else
            Current_Task.Status := PREEMPTED_MANDATORY ;
        end if ;

        V_XTasking.Set_Priority ( Preempted_Priority,
                                  Current_Task.Task_ID ) ;
        Current_Task.Time_Remaining := Current_Task.Time_Remaining -
                               Duration ( Float ( Calendar.Clock -
                                                  Current_Task.Started_At )) ;
        --
        --    Due to the time slicing on a multiuser UNIX system, it is
        -- likely that our predicted time remaining will be wrong.  This is
        -- due to our not having the full processor utilization.  This IF
        -- clause prevents the time-remaining from going negative.
        --
        if Current_Task.Time_Remaining < 0.0 then
            Current_Task.Time_Remaining := 0.1 ;
        end if ;
        Current_Task.Latest_Start_Time := Current_Task.Deadline -
                                          Current_Task.Time_Remaining ;
        Deadline_Ordered_Queue.Add ( Current_Task,
                                     Deadline_Que,
                                     Deadline_Queue_Back ) ;
        Latest_Start_Time_Ordered_Queue.Add ( Current_Task,
                                              Latest_Start_Time_Que,
                                              Latest_Start_Time_Queue_Back ) ;
        New_Current_Task.Started_At := Calendar.Clock ;
        if New_Current_Task.Status = READY or
           New_Current_Task.Status = PREEMPTED_MANDATORY then
            New_Current_Task.Status := EXECUTING_MANDATORY ;
        else
            New_Current_Task.Status := EXECUTING_OPTIONAL ;
        end if;
        V_XTasking.Set_Priority ( Executing_Priority,
                                  New_Current_Task.Task_ID ) ;

        Current_Task := New_Current_Task ;
    end Preempt_Current_Task ;

    ------------------------------------------------------------
    --                                                        --
```

```
--         FUNCTION   Delay_Till_Next_Action          --
--                                                    --
------------------------------------------------------
function Delay_Till_Next_Action return duration is
    The_Delay  : Duration := 0.01 ;
    Temp_Delay : Duration ;
begin
    if not Start_Time_Ordered_Queue.Is_Empty ( Ready_Que ) then
        The_Delay := Start_Time_Ordered_Queue.Front_Of
                          ( Ready_Que ).Start_Time - Calendar.Clock ;
    end if ;
    if not Latest_Start_Time_Ordered_Queue.
           Is_Empty(Latest_Start_Time_Que) then
        Temp_Delay := Latest_Start_Time_Ordered_Queue.Front_Of
                          ( Latest_Start_Time_Que ).Latest_Start_Time
                      - Calendar.Clock ;
        if Temp_Delay < The_Delay then
            return Temp_Delay ;
        else
            return The_Delay ;
        end if ;
    else
        return The_Delay ;
    end if ;

end Delay_Till_Next_Action ;

------------------------------------------------------
--                                                  --
--              PROCEDURE   Start_Up_Tasks          --
--                                                  --
------------------------------------------------------
procedure Start_Up_Tasks is
begin
    The_TCB := Start_Time_Ordered_Queue.Front_Of( Ready_Que ) ;
    while The_TCB.Start_Time <= Calendar.Clock loop
        Start_Time_Ordered_Queue.Pop ( Ready_Que, Ready_Queue_Front ) ;

        case The_TCB.Task_Kind is

            when PERIODIC =>
                --
                --    First check to see if it is time to stop this task.
                --
                if The_TCB.Deadline > Calendar.Clock then
                    --
                    --    The Deadline/Stop_Time for this periodic task
                    -- has not come yet so, resume the task and set its
                    -- priority.  Note that the Resume_Task call is in
                    -- a BEGIN - EXECPTION - END structure to capture an
                    -- invalid resume that signals a missed deadline.
                    --
                    begin
                        V_XTasking.Resume_Task ( The_TCB.Task_ID ) ;
                        V_XTasking.Set_Priority ( The_TCB.Priority,
                                                  The_TCB.Task_ID ) ;
                        The_TCB.Start_Time := The_TCB.Start_Time
                                + The_TCB.Period ;
                        The_TCB.Status := EXECUTING ;
                    exception
                        when V_XTasking.Invalid_Resume =>
                            --
                            -- Missed the period so restart the periods with
                            -- this time and notify the reasoning process.
                            --
                            The_TCB.Start_Time := Calendar.Clock
                                                + The_TCB.Period ;
                            Reasoning_Process.Missed_Deadline
                                                ( The_TCB.Integer_Task_ID,
                                                  Calendar.Clock ) ;
                    end ;
                    --
                    --    Place the task back in the ready queue so it
                    -- will be executed again.
                    --
                    Start_Time_Ordered_Queue.Add ( The_TCB,
                                                   Ready_Que,
                                                   Ready_Queue_Back ) ;
                else
                    --
                    --    In this case, it is time to stop this periodic
                    -- task.  So the Un_Schedule routine is used.
                    --
                    Un_Schedule ( The_TC3 ) ;
                end if ;

            when SINGULAR | ANY_TIME =>

                The_TCB.Time_Remaining :=
                      Duration ( Float ( The_TCB.Mandatory_Duration )
                                 / ( 1.0 - Periodic_Utilization_Budget )) ;
                case The_TCB.Task_Kind is

                    when ANY_TIME =>
                        The_TCB.Any_Time_Variables.Continue := TRUE ;
                        Any_Time_Tasks.Store_Variables
                                          ( The_TCB.Any_Time_Variables,
                                            The_TCB.Task_ID ) ;
                    when SINGULAR =>
                        The_TCB.Singular_Variables.Continue := TRUE ;
                        Singular_Tasks.Store_Variables
                                          ( The_TCB.Singular_Variables,
                                            The_TCB.Task_ID ) ;

                     when OTHERS =>
                        null ;

                end case ;

                V_XTasking.Resume_Task ( The_TCB.Task_ID ) ;
                --
                --    The first if clause checks to see if there is a
                -- task that is currently executing at the highest
                -- Non-Periodic priority.  If there is not, then the
                -- task we are starting is given the highest priority
                -- and started.
                --
                if Current_Task = Null then
                    The_TCB.Status := EXECUTING_MANDATORY ;
                    The_TCB.Started_At := Calendar.Clock ;
                    V_XTasking.Set_Priority ( Executing_Priority,
                                              The_TCB.Task_ID ) ;
                    Current_Task := The_TCB ;
                --
                --    We get here if there already is a task scheduled and
                -- running.  In this case we follow the earliest deadline
                -- first algorithm.
                --
                elsif Current_Task.Deadline < The_TCB.Deadline then
                    The_TCB.Status := PREEMPTED_MANDATORY ;
                    V_XTasking.Set_Priority ( Preempted_Priority,
                                              The_TCB.Task_ID ) ;
                    Deadline_Ordered_Queue.Add ( The_TCB,
                                                 Deadline_Que,
                                                 Deadline_Queue_Back ) ;
                    Latest_Start_Time_Ordered_Queue.Add
                                        ( The_TCB,
                                          Latest_Start_Time_Que,
                                          Latest_Start_Time_Queue_Back ) ;
                --
                --    This condition is reached if the new task's deadline
                -- is before the currently executing task's and the current-
                -- ly executing task was not scheduled because it
                -- reached its Latest_Start_Time.
```

```
                --
                elsif Preempted_Value = integer'last then
                    Preempt_Current_Task ( The_TCB ) ;
                    --
                    --    Okay, now someone might miss a deadline.  We
                    -- reach this point when the current task's deadline is
                    -- earlier than the new task's, and the current
                    -- task was scheduled because it reached its
                    -- latest start time.  So the task with
                    --
                elsif Current_Task.Importance < Preempted_Value then
                    --
                    --    This looks backwards but lower values are of
                    -- a higher importance.  Note that the current task
                    -- has been scheduled because it reached its latest
                    -- start time and thus will miss its deadline if
                    -- preempted.  Preempted_Value would be zero if the
                    -- task was not scheduled because it reached its
                    -- latest start time.  If the current task is an
                    -- optional part than it is discarded and the new
                    -- one is scheduled in its place.
                    --
                    if Current_Task.Status = EXECUTING_MANDATORY then
                        Reasoning_Process.Missed_Deadline
                                        ( The_TCB.Integer_Task_ID,
                                          Calendar.Clock ) ;
                    end if ;
                    Un_Schedule ( Current_Task ) ;
                    The_TCB.Status := EXECUTING_MANDATORY ;
                    The_TCB.Started_At := Calendar.Clock ;
                    V_XTasking.Set_Priority ( Executing_Priority,
                                              The_TCB.Task_ID ) ;
                    Current_Task := The_TCB ;
                    --
                    --    In this case, the new task will not be scheduled
                    -- before its deadline so the reasoning process sent a
                    -- message that the task missed its deadline and the
                    -- task is discarded.
                    --
                else
                    Un_Schedule ( The_TCB ) ;
                    Reasoning_Process.Missed_Deadline
                                        ( The_TCB.Integer_Task_ID,
                                          Calendar.Clock ) ;
                end if ;

            end case ;

            if Start_Time_Ordered_Queue.Is_Empty ( Ready_Que ) then
                exit ;
            end if ;

            The_TCB := Start_Time_Ordered_Queue.Front_Of( Ready_Que ) ;
        end loop ;
    end Start_Up_Tasks ;

    ------------------------------------------------------------
    --                                                        --
    --       PROCEDURE Schedule_Non_Periodic_Tasks            --
    --                                                        --
    ------------------------------------------------------------
    --
    --    This procedure should be called by an interrupt entry into the
    -- task caused by a timer.  The interrupt would signal the end of the
    -- time a task can wait to start executing before it will miss its
    -- deadline, i.e. latest_start_time .
    --
    procedure Schedule_Non_Periodic_Tasks is
    begin
        The_TCB := Latest_Start_Time_Ordered_Queue.
                            Front_Of( Latest_Start_Time_Que ) ;
        while The_TCB.Latest_Start_Time <= Calendar.Clock loop
```

```
            Current_Task.Time_Remaining :=
                        Current_Task.Time_Remaining -
                        Duration ( Float ( Calendar.Clock -
                                           Current_Task.Started_At )) ;
            --
            --    Due to the time slicing on a multiuser UNIX system, it is
            -- likely that our predicted time remaining will be wrong.  This is
            -- due to our not having the full processor utilization.  This IF
            -- clause prevents the time-remaining from going negative.
            --
            if Current_Task.Time_Remaining < 0.0 then
                Current_Task.Time_Remaining := 0.1 ;
            end if ;
            --
            --    The current strategy is to always try and complete at least
            -- the mandatory part of a task.  This implies that a task that
            -- has reached its latest start time and has not completed its
            -- mandatory part will always preempt a task that is executing its
            -- optional part.
            --
            if Current_Task.Status = EXECUTING_OPTIONAL and
               The_TCB.Status = PREEMPTED_MANDATORY then
                Deadline_Ordered_Queue.Remove_Item
                                        ( Deadline_Que,
                                          Deadline_Ordered_Queue.Position_Of
                                                ( The_TCB, Deadline_Que ));
                Latest_Start_Time_Ordered_Queue.Pop
                                        ( Latest_Start_Time_Que,
                                          Latest_Start_Time_Queue_Front ) ;
                Preempt_Current_Task ( The_TCB ) ;
            --
            --   If we get here we then we favor a mandatory part over an
            -- optional part and the TCB at the head of the LST queue will be
            -- unscheduled.
            --
            elsif Current_Task.Status = EXECUTING_MANDATORY and
                  The_TCB.Status = PREEMPTED_OPTIONAL then
                Un_Schedule ( The_TCB ) ;
            --
            --    If we get here then both tasks are executing the same parts
            -- either both mandatory of both optional.  This checks to see if
            -- there is any slack time in the currently executing task and
            -- preempts it if there is.
            --
            elsif ( Current_Task.Time_Remaining >
                    Current_Task.Deadline - Calendar.Clock ) then
                Deadline_Ordered_Queue.Remove_Item
                                        ( Deadline_Que,
                                          Deadline_Ordered_Queue.Position_Of
                                                ( The_TCB, Deadline_Que ));
                Latest_Start_Time_Ordered_Queue.Pop
                                        ( Latest_Start_Time_Que,
                                          Latest_Start_Time_Queue_Front ) ;
                Preempt_Current_Task ( The_TCB ) ;
            --
            --    If we make it here then there is no slack time. This part
            -- allows the task with the highest importance to continue.
            -- The other task is aborted and will miss its deadline.
            --
            elsif The_TCB.Importance < Preempted_Value then
                Preempted_Value := The_TCB.Importance ;
                Un_Schedule ( Current_Task ) ;
                Reasoning_Process.Missed_Deadline ( Current_Task.Integer_Task_ID,
                                                    Calendar.Clock ) ;
                Current_Task := The_TCB ;
            else
                Un_Schedule ( The_TCB ) ;
                Reasoning_Process.Missed_Deadline ( The_TCB.Integer_Task_ID,
                                                    Calendar.Clock ) ;
            end if ;
```

```
            if not Latest_Start_Time_Ordered_Queue.
                                        Is_Empty( Latest_Start_Time_Que ) then
                exit ;
            else
                The_TCB := Latest_Start_Time_Ordered_Queue.
                                        Front_Of( Latest_Start_Time_Que ) ;
            end if ;

        end loop ;

    end Schedule_Non_Periodic_Tasks ;

begin

    if Not Start_Time_Ordered_Queue.Is_Empty ( Ready_Que ) then
        Start_Up_Tasks ;
    end if ;

    if not Latest_Start_Time_Ordered_Queue.
                        Is_Empty( Latest_Start_Time_Que ) then
        Schedule_Non_Periodic_Tasks ;
    end if ;

    return Delay_Till_Next_Action ;

end Dispatch_Tasks ;
```

```
separate ( Task_Manager_Package )
    ------------------------------------------------------------
    --                                                        --
    --              FUNCTION   Feasible                       --
    --                                                        --
    ------------------------------------------------------------

--
--   This procedure performs a rate mortonic scheduleability test on any
-- newly added periodic task to insure that it can meet its deadline before it
-- is actually started.  Any error is signalled back to the 'Reasoning Process'
-- by the procedure that calls this predicate and gets a 'false' answer.
--
--   In the case of an any_time or singular task, the feasiblity check consists
-- of determining if enough time exists to complete at least the mandatory
-- part of the task before its deadline.  As of now the effect of periodic
-- tasks is included in the feasibility test but the effect of other any_time
-- or singular tasks is not.
--
function Feasible ( The_TCB : in Task_Control_Block_Ptr ) return boolean is

    Min_Task_Duration,
    Additional_Utilization : Float ;

begin
    case The_TCB.Task_Kind is
        when PERIODIC =>
            if Periodic_Utilization <= Required_Utilization then
                return True ;
            else
                return False ;
            end if ;

        when SINGULAR | ANY_TIME =>
            Min_Task_Duration :=  Float (The_TCB.Mandatory_Duration)
                                / (1.0 - Periodic_Utilization ) ;
            if (float (The_TCB.Deadline - Calendar.Clock) >
                Min_Task_Duration) and then
               (float (The_TCB.Deadline - The_TCB.Start_Time) >
                Min_Task_Duration) then
                return True ;
            else
                return False ;
            end if ;

    end case ;

end Feasible ;
```

```
separate ( Task_Manager_Package )
    ------------------------------------------------------
    --                                                  --
    --              FUNCTION   Find_TCB                 --
    --                                                  --
    ------------------------------------------------------

--
--    The purpose of this function is to find the TCB for a task given
-- the task's Task_ID.  Currently, the TCBs are stored in a link-list
-- structure so this is an O(n) operation.  Changing to a binary tree or
-- some other structure could reduce this search to O(log n).
--
function Find_TCB ( The_Task_ID : in Integer ) return Task_Control_Block_Ptr is

    Found_TCB : Task_Control_Block_Ptr ;

    Procedure Search_For_Task ( The_TCB  : in Task_Control_Block_Ptr ;
                                Continue : out Boolean ) is
    begin
        if The_TCB.Integer_Task_ID = The_Task_ID then
            Found_TCB  := The_TCB ;
            Continue := False ;
        else
            Continue := True ;
        end if ;
    end Search_For_Task ;
    procedure Traverse_Task_List is new Task_ID_Queue.Iterate
                                            ( Process => Search_For_Task ) ;
begin
    Traverse_Task_List ( Tasks_By_ID ) ;
    return Found_TCB ;
end Find_TCB ;
```

```
separate ( Task_Manager_Package )
    ------------------------------------------------------
    --                                                  --
    --                FUNCTION Get_TCB                  --
    --                                                  --
    ------------------------------------------------------

--
--    This procedure takes a Task_Control_Block as input and returns a pointer
-- to a partially filled out Task_Control_Block.  The Items not filled out
-- are the assigned priority, the start time, and the mode.  Each of these
-- values is filled in by the SCHEDULE procedure based upon the current
-- scheduling scheme that is in place.
--
function Get_TCB ( Display_Flag   : in Boolean ;
                   The_Type       : in Task_Kind_Type ;
                   The_Kind       : in Integer ;
                   The_Period     : in Duration ;
                   The_Deadline   : in Calendar.Time ;
                   The_Start_Time : in Calendar.Time ;
                   The_Importance : in Integer )
                                          return Task_Control_Block_Ptr is

    New_TCB_Ptr  : Task_Control_Block_Ptr ;

begin
    --
    -- First objective is to get a Task_Control_Block for the specific
    -- task that you are trying to instantiate.  The TCB can either come
    -- from the free list or a new one is created.
    --
    if Free_Task_List ( The_Type ) = null then
        New_TCB_Ptr := new Task_Control_Block_Type ( The_Type ) ;
        New_TCB_Ptr.Period     := The_Period ;
        New_TCB_Ptr.Task_Kind  := The_Type ;
        New_TCB_Ptr.Deadline   := The_Deadline ;
        New_TCB_Ptr.Start_Time := The_Start_Time ;
        New_TCB_Ptr.Importance := The_Importance ;
        case The_Type is

            when ANY_TIME =>

                New_TCB_Ptr.Any_Time_Variables.Kind     := The_Kind ;
                New_TCB_Ptr.Any_Time_Variables.Display  := Display_Flag ;
                New_TCB_Ptr.Any_Time_Variables.Continue := TRUE ;

                Any_Time_Tasks.Create ( New_TCB_Ptr.Any_Time_Variables,
                                        New_TCB_Ptr.The_Any_Time_Task_Ptr,
                                        New_TCB_Ptr.Task_ID,
                                        New_TCB_Ptr.Mandatory_Duration,
                                        New_TCB_Ptr.Optional_Duration ) ;
            when PERIODIC =>

                New_TCB_Ptr.Periodic_Variables.Kind     := The_Kind ;
                New_TCB_Ptr.Periodic_Variables.Display  := Display_Flag ;
                New_TCB_Ptr.Periodic_Variables.Continue := TRUE ;
                Periodic_Tasks.Create ( New_TCB_Ptr.Periodic_Variables,
                                        New_TCB_Ptr.The_Periodic_Task_Ptr,
                                        New_TCB_Ptr.Task_ID,
                                        New_TCB_Ptr.Mandatory_Duration,
                                        New_TCB_Ptr.Optional_Duration ) ;
            when SINGULAR =>

                New_TCB_Ptr.Singular_Variables.Kind     := The_Kind ;
                New_TCB_Ptr.Singular_Variables.Display  := Display_Flag ;
                New_TCB_Ptr.Singular_Variables.Continue := TRUE ;

                Singular_Tasks.Create ( New_TCB_Ptr.Singular_Variables,
                                        New_TCB_Ptr.The_Singular_Task_Ptr,
                                        New_TCB_Ptr.Task_ID,
                                        New_TCB_Ptr.Mandatory_Duration,
                                        New_TCB_Ptr.Optional_Duration ) ;
```

```
    end case ;
else
    --
    --    A currently used task of this type is on the Free_Task_List so
    -- use it.  That means we need to change the variables for the task
    -- to the new values that are contained it the TCB passed in.  In
    -- addition, the task itself needs to provide the durations of the
    -- new job.  Since all non-running tasks are assumed to be in the
    -- suspended state, first the task is resumed, and then a call to
    -- the tasks change variables is made.  Note the the task itself
    -- will place itself back into the suspended state.
    --
    New_TCB_Ptr := Free_Task_List ( The_Type ) ;
    Free_Task_List ( The_Type ) := New_TCB_Ptr.Next ;
    New_TCB_Ptr.Period     := The_Period ;
    New_TCB_Ptr.Task_Kind  := The_Type ;
    New_TCB_Ptr.DeadLine   := The_Deadline ;
    New_TCB_Ptr.Start_Time := The_Start_Time ;
    New_TCB_Ptr.Importance := The_Importance ;
    case The_Type is

        when ANY_TIME =>

            New_TCB_Ptr.Any_Time_Variables.Kind     := The_Kind ;
            New_TCB_Ptr.Any_Time_Variables.Display  := Display_Flag ;
            New_TCB_Ptr.Any_Time_Variables.Continue := TRUE ;
            V_XTasking.Resume_Task ( New_TCB_Ptr.Task_ID ) ;
            New_TCB_Ptr.The_Any_Time_Task_Ptr.Change_Variables
                                    ( New_TCB_Ptr.Any_Time_Variables,
                                      New_TCB_Ptr.Mandatory_Duration,
                                      New_TCB_Ptr.Optional_Duration ) ;
        when PERIODIC =>

            New_TCB_Ptr.Periodic_Variables.Kind     := The_Kind ;
            New_TCB_Ptr.Periodic_Variables.Display := Display_Flag ;
            New_TCB_Ptr.Periodic_Variables.Continue := TRUE ;
            V_XTasking.Resume_Task ( New_TCB_Ptr.Task_ID ) ;
            New_TCB_Ptr.The_Periodic_Task_Ptr.Change_Variables
                                    ( New_TCB_Ptr.Periodic_Variables,
                                      New_TCB_Ptr.Mandatory_Duration,
                                      New_TCB_Ptr.Optional_Duration ) ;
        when SINGULAR =>

            New_TCB_Ptr.Singular_Variables.Kind     := The_Kind ;
            New_TCB_Ptr.Singular_Variables.Display  := Display_Flag ;
            New_TCB_Ptr.Singular_Variables.Continue := TRUE ;
            V_XTasking.Resume_Task ( New_TCB_Ptr.Task_ID ) ;
            New_TCB_Ptr.The_Singular_Task_Ptr.Change_Variables
                                    ( New_TCB_Ptr.Singular_Variables,
                                      New_TCB_Ptr.Mandatory_Duration,
                                      New_TCB_Ptr.Optional_Duration ) ;
    end case ;
end if ;
--
-- Now we need to update the values used to control task scheduling
--
case The_Type is

    when ANY_TIME | SINGULAR =>

        Num_of_NP_Tasks := Num_of_NP_Tasks + 1 ;

    when PERIODIC =>

        Number_of_Periodic_Tasks := Number_of_Periodic_Tasks + 1 ;
        --
        --    This is where the min and max utilizations are calculated.
        -- Other utilizations based upon changing periods could be added.
        --
        Mandatory_Utilization := Mandatory_Utilization +
                                 ( Float ( New_TCB_Ptr.Mandatory_Duration )
                                   + Twice_Task_Swap_Delay )
                                 / Float ( New_TCB_Ptr.Period ) ;
        Optional_Utilization := Optional_Utilization +
                                ( Float ( New_TCB_Ptr.Mandatory_Duration )
                                  + Float ( New_TCB_Ptr.Optional_Duration )
                                  + Twice_Task_Swap_Delay )
                                / Float ( New_TCB_Ptr.Period ) ;
        --
        --    The required utilization is the rate-montonic theory value
        -- that is needed for the N tasks to be feasible.  Note that the
        -- value is adjusted to be a percentage of the entire utilzation
        -- based upon the Periodic_Utilization_Budget.
        --
        N := float (Number_of_Periodic_Tasks) ;
        Required_Utilization := N * ( 2.0 ** (1.0 / N) -1.0 ) *
                                Periodic_Utilization_Budget ;
        --
        --    The new task is now added to the data structures that keep
        -- track of periodic tasks.
        --
        Period_Ordered_Queue.Add ( New_TCB_Ptr,
                                   Tasks_By_Period,
                                   Period_Queue_Back ) ;
        Importance_Ordered_Queue.Add ( New_TCB_Ptr,
                                       Periodic_Importance_Que,
                                       Importance_Queue_Back ) ;
        --
        -- The goal here is to determine the size of each priority
        -- bin.  That means that given some number of priorities
        -- and a set of tasks whose cardinality is larger than the
        -- number of priorities and each with an unrestrained period,
        -- what should the range of periods for each available
        -- priority be?  This is only calculated when there are more
        -- tasks than priorities to save a little on overhead.
        --
        if Number_of_Periodic_Tasks > Number_of_Periodic_Priorities then
           Periodic_Bin_Size := Integer ( Number_of_Periodic_Tasks /
                                          Number_of_Periodic_Priorities ) ;
        end if ;
    end case ;

    --
    --   Add the new task to the queue used to match an integer Task_ID to a
    -- TCB. This is necessary for communciations between the RP and TM
    -- because a TCB is a private type.
    --
    New_TCB_Ptr.Integer_Task_ID := Task_ID_To_Integer( New_TCB_Ptr.Task_ID ) ;
    Task_ID_Queue.Add ( New_TCB_Ptr, Tasks_By_ID, Tasks_By_ID_Queue_Back ) ;

    return New_TCB_Ptr ;

end Get_TCB ;
```

```
separate ( Task_Manager_Package )
    ------------------------------------------------------------
    --                                                        --
    --                      FUNCTION Modify                   --
    --                                                        --
    ------------------------------------------------------------

--
--    The problem with modifing a task is first figuring out what has
-- been modified and then making the appropriate changes to both the
-- variables used for scheduling of the task and the variables used by
-- the task for its execution.
--
procedure Modify ( The_TCB         : in Task_Control_Block_Ptr;
                   New_Period      : Duration ;
                   New_Display     : Boolean ;
                   New_Deadline    : Calendar.Time ;
                   New_Start_Time  : Calendar.Time ;
                   New_Importance  : Integer ) is

    Change_Task_Variables,
    Change_Schedule_Variables,
    Re_Schedule_Periodics,
    Re_Schedule_Non_Periodics     : Boolean := FALSE ;

begin
    case The_TCB.Task_Kind is

        when PERIODIC =>

            Re_Schedule_Periodics := FALSE ;

            if The_TCB.Periodic_Variables.Display /= New_Display then
               The_TCB.Periodic_Variables.Display := New_Display ;
               Periodic_Tasks.Store_Variables ( The_TCB.Periodic_Variables,
                                                The_TCB.Task_ID ) ;
            end if ;

            if The_TCB.Period /= New_Period then
                Change_Schedule_Variables := True ;
                Mandatory_Utilization := Mandatory_Utilization
                                - ( Float ( The_TCB.Mandatory_Duration )
                                          + Twice_Task_Swap_Delay )
                                  / Float ( The_TCB.Period )
                                + ( Float ( The_TCB.Mandatory_Duration )
                                          + Twice_Task_Swap_Delay )
                                  / Float ( New_Period ) ;
                Optional_Utilization := Optional_Utilization
                                - ( Float ( The_TCB.Mandatory_Duration )
                                        + Float ( The_TCB.Optional_Duration )
                                        + Twice_Task_Swap_Delay )
                                  / Float ( The_TCB.Period )
                                + ( Float ( The_TCB.Mandatory_Duration )
                                        + Float ( The_TCB.Optional_Duration )
                                        + Twice_Task_Swap_Delay )
                                  / Float ( New_Period );
                The_TCB.Period := New_Period ;
                --
                --    It is necessary here to remove the TCB from the queue and
                -- and then reinsert in it the correct order.  Just changing
                -- the period by itself does not change the tasks position in
                -- the queue.
                --
                Period_Ordered_Queue.Remove_Item
                             ( Tasks_By_Period, Period_Ordered_Queue.
                                   Position_Of ( The_TCB, Tasks_By_Period )) ;
                Period_Ordered_Queue.Add ( The_TCB, Tasks_By_Period,
                                           Period_Queue_Back ) ;
                Re_Schedule_Periodics := True ;
            end if ;

            if The_TCB.Importance /= New_Importance then
                The_TCB.Importance := New_Importance ;
                --
                --    It is necessary here to remove the TCB from the queue and
                -- and then reinsert in it the correct order.  Just changing
                -- the period by itself does not change the tasks position in
                -- the queue.
                --
                Importance_Ordered_Queue.Remove_Item
                           ( Periodic_Importance_Que,
                             Importance_Ordered_Queue.Position_Of
                                        (The_TCB, Periodic_Importance_Que)) ;
                Importance_Ordered_Queue.Add ( The_TCB, Periodic_Importance_Que,
                                               Importance_Queue_Back ) ;
                Re_Schedule_Periodics := True ;
            end if ;

            if The_TCB.Deadline /= New_Deadline then
                The_TCB.Deadline := New_Deadline ;
            end if ;

            if The_TCB.Status = READY then
                if The_TCB.Start_Time /= New_Start_Time then
                    The_TCB.Start_Time := New_Start_Time ;
                    Start_Time_Ordered_Queue.Remove_Item ( Ready_Que,
                           Start_Time_Ordered_Queue.Position_Of ( The_TCB,
                                                                  Ready_Que )) ;
                    Start_Time_Ordered_Queue.Add ( The_TCB, Ready_Que,
                                                   Ready_Queue_Back ) ;
                end if ;
            end if ;

            if Re_Schedule_Periodics then
                Assign_Periodic_Priorities ;
            end if ;

        when ANY_TIME | SINGULAR =>

            if The_TCB.Any_Time_Variables.Display /= New_Display then
               The_TCB.Any_Time_Variables.Display := New_Display ;
               Any_Time_Tasks.Store_Variables ( The_TCB.Any_Time_Variables,
                                                The_TCB.Task_ID ) ;
            end if ;

            case The_TCB.Status is

                when DISCARDED | COMPLETED =>

                    null ;

                when READY =>

                    if The_TCB.Start_Time /= New_Start_Time then
                        The_TCB.Start_Time := New_Start_Time ;
                        Start_Time_Ordered_Queue.Remove_Item ( Ready_Que,
                           Start_Time_Ordered_Queue.Position_Of ( The_TCB,
                                                                  Ready_Que )) ;
                        Start_Time_Ordered_Queue.Add ( The_TCB, Ready_Que,
                                                       Ready_Queue_Back ) ;
                    end if ;

                    if The_TCB.Deadline /= New_Deadline then
                        The_TCB.Deadline := New_Deadline ;
                    end if ;

                    if The_TCB.Importance /= New_Importance then
                        The_TCB.Importance := New_Importance ;
                    end if ;

                when EXECUTING_MANDATORY | EXECUTING_OPTIONAL |
                     PREEMPTED_MANDATORY | PREEMPTED_OPTIONAL =>

                    --
```

```
        --    The deadline and importance can be changed but
        -- the change will not affect the Current_Task
        -- immediately.  It will alter the queues, but only
        -- dispatch_tasks or Completed_Non_Periodic can place
        -- a new task as the current task.
        --

        if The_TCB.Deadline /= New_Deadline then
            The_TCB.Deadline := New_Deadline ;

            if The_TCB.Status = EXECUTING_MANDATORY or
              The_TCB.Status = EXECUTING_OPTIONAL  then
                The_TCB.Time_Remaining :=
                      The_TCB.Time_Remaining -
                      Duration ( Float ( Calendar.Clock -
                                         The_TCB.Started_At )) ;
            else

                The_TCB.Latest_Start_Time := The_TCB.Deadline -
                                             The_TCB.Time_Remaining ;
                Deadline_Ordered_Queue.Remove_Item
                            ( Deadline_Que,
                              Deadline_Ordered_Queue.Position_Of
                                ( The_TCB, Deadline_Que ));
                Latest_Start_Time_Ordered_Queue.Remove_Item
                      ( Latest_Start_Time_Que,
                        Latest_Start_Time_Ordered_Queue.Position_Of
                              ( The_TCB, Latest_Start_Time_Que ));
                Deadline_Ordered_Queue.Add ( The_TCB,
                                             Deadline_Que,
                                             Deadline_Queue_Back ) ;
                Latest_Start_Time_Ordered_Queue.Add ( The_TCB,
                                Latest_Start_Time_Que,
                                Latest_Start_Time_Queue_Back ) ;
            end if ;
        end if ;

        if The_TCB.Importance /= New_Importance then
            The_TCB.Importance := New_Importance ;
        end if ;

    when OTHERS =>
        null ;

  end case ;

end case ;

end Modify ;
```

```
separate ( Task_Manager_Package )
    ------------------------------------------------------------
    --                                                        --
    --          PROCEDURE  Non_Periodic_Completed             --
    --                                                        --
    ------------------------------------------------------------

--
--    Note that only SINGULAR or ANY_TIME tasks can call this procedure
--

procedure Non_Periodic_Completed ( The_TCB : in Task_Control_Block_Ptr ) is

    Need_New_Current_Task : Boolean := False ;

begin

    case The_TCB.Status is

        when DISCARDED =>

            The_TCB.Status := COMPLETED ;
            The_TCB.Next :=  Free_Task_List ( The_TCB.Task_Kind ) ;
            Free_Task_List ( The_TCB.Task_Kind ) := The_TCB ;

        when EXECUTING_MANDATORY | PREEMPTED_MANDATORY =>

            The_TCB.Status := PREEMPTED_OPTIONAL ;
            V_XTasking.Set_Priority ( Preempted_Priority, The_TCB.Task_ID ) ;

            if The_TCB.Status = PREEMPTED_MANDATORY then
                Deadline_Ordered_Queue.Remove_Item
                                        ( Deadline_Que,
                                          Deadline_Ordered_Queue.Position_Of
                                                ( The_TCB, Deadline_Que ));
                Latest_Start_Time_Ordered_Queue.Remove_Item
                                    ( Latest_Start_Time_Que,
                                      Latest_Start_Time_Ordered_Queue.Position_Of
                                            ( The_TCB, Latest_Start_Time_Que ));
            else
                Need_New_Current_Task := TRUE ;
                Preempted_Value := Integer'Last ;
            end if ;

            The_TCB.Time_Remaining :=
                              Duration ( Float ( The_TCB.Optional_Duration )
                                       / ( 1.0 - Periodic_Utilization_Budget )) ;
            The_TCB.Latest_Start_Time := The_TCB.Deadline -
                                         The_TCB.Time_Remaining ;
            Deadline_Ordered_Queue.Add ( The_TCB,
                                         Deadline_Que,
                                         Deadline_Queue_Back ) ;
            Latest_Start_Time_Ordered_Queue.Add
                                          ( The_TCB,
                                            Latest_Start_Time_Que,
                                            Latest_Start_Time_Queue_Back ) ;

        when EXECUTING_OPTIONAL =>

            Preempted_Value := Integer'Last ;
            Need_New_Current_Task := TRUE ;
            --
            --    A singular task is completed and can be placed back in the
            -- free task list.  An any time task may have time for another
            -- execution cycle.
            --
            if The_TCB.Task_Kind = SINGULAR then
                The_TCB.Singular_Variables.Continue := FALSE ;
                Singular_Tasks.Store_Variables ( The_TCB.Singular_Variables,
                                                 The_TCB.Task_ID ) ;
                The_TCB.Status := COMPLETED ;
                The_TCB.Next :=  Free_Task_List ( The_TCB.Task_Kind ) ;
```

```
            Free_Task_List ( The_TCB.Task_Kind ) := The_TCB ;
        else
            The_TCB.Status := PREEMPTED_OPTIONAL ;
            V_XTasking.Set_Priority ( Preempted_Priority,
                                      The_TCB.Task_ID ) ;
            --
            --    Now we have to check if there is enough time remaining
            -- to execute another loop through the Any Time task.  If there
            -- is then put the task back in the queues.  If there isn't
            -- then put the task in the free task list.
            --
            The_TCB.Time_Remaining :=
                            Duration ( Float ( The_TCB.Optional_Duration )
                                     / ( 1.0 - Periodic_Utilization_Budget )) ;
            The_TCB.Latest_Start_Time := The_TCB.Deadline -
                                         The_TCB.Time_Remaining ;
            if The_TCB.Latest_Start_Time > Calendar.Clock then
                Deadline_Ordered_Queue.Add ( The_TCB,
                                             Deadline_Que,
                                             Deadline_Queue_Back ) ;
                Latest_Start_Time_Ordered_Queue.Add
                                           ( The_TCB,
                                             Latest_Start_Time_Que,
                                             Latest_Start_Time_Queue_Back ) ;
            else
                The_TCB.Any_Time_Variables.Continue := FALSE ;
                Any_Time_Tasks.Store_Variables ( The_TCB.Any_Time_Variables,
                                                 The_TCB.Task_ID ) ;
                The_TCB.Status := COMPLETED ;
                The_TCB.Next :=  Free_Task_List ( The_TCB.Task_Kind ) ;
                Free_Task_List ( The_TCB.Task_Kind ) := The_TCB ;
            end if ;
        end if ;

    when PREEMPTED_OPTIONAL =>
        --
        --    Since the completed tasks state is PREEMPTED_OPTIONAL,
        -- it is in both the deadline and latest start time queues.  The
        -- first jod is to remove it.
        --
        Deadline_Ordered_Queue.Remove_Item
                                ( Deadline_Que,
                                  Deadline_Ordered_Queue.Position_Of
                                             ( The_TCB, Deadline_Que ));
        Latest_Start_Time_Ordered_Queue.Remove_Item
                             ( Latest_Start_Time_Que,
                               Latest_Start_Time_Ordered_Queue.Position_Of
                                      ( The_TCB, Latest_Start_Time_Que ));

        --
        --    A singular task is completed and can be placed back in the
        -- free task list.  An any time task may have time for another
        -- execution cycle.
        --
        if The_TCB.Task_Kind = SINGULAR then
            The_TCB.Singular_Variables.Continue := FALSE ;
            Singular_Tasks.Store_Variables ( The_TCB.Singular_Variables,
                                             The_TCB.Task_ID ) ;
            The_TCB.Status := COMPLETED ;
            The_TCB.Next :=  Free_Task_List ( The_TCB.Task_Kind ) ;
            Free_Task_List ( The_TCB.Task_Kind ) := The_TCB ;
        else
            The_TCB.Time_Remaining :=
                            Duration ( Float ( The_TCB.Optional_Duration )
                                     / ( 1.0 - Periodic_Utilization_Budget )) ;
            The_TCB.Latest_Start_Time := The_TCB.Deadline -
                                         The_TCB.Time_Remaining ;
            if The_TCB.Latest_Start_Time > Calendar.Clock then
                Deadline_Ordered_Queue.Add ( The_TCB,
                                             Deadline_Que,
                                             Deadline_Queue_Back ) ;
                Latest_Start_Time_Ordered_Queue.Add
                                           ( The_TCB,
                                             Latest_Start_Time_Que,
                                             Latest_Start_Time_Queue_Back ) ;
            else
                The_TCB.Any_Time_Variables.Continue := FALSE ;
                Any_Time_Tasks.Store_Variables ( The_TCB.Any_Time_Variables,
                                                 The_TCB.Task_ID ) ;
                The_TCB.Status := COMPLETED ;
                The_TCB.Next :=  Free_Task_List ( The_TCB.Task_Kind ) ;
                Free_Task_List ( The_TCB.Task_Kind ) := The_TCB ;
            end if ;

        when OTHERS =>
            null;

    end case ;

    if Need_New_Current_Task and
       not Deadline_Ordered_Queue.Is_Empty ( Deadline_Que ) then
        Current_Task := Deadline_Ordered_Queue.Front_Of ( Deadline_Que ) ;
        Deadline_Ordered_Queue.Pop ( Deadline_Que,
                                     Deadline_Queue_Front) ;
        Latest_Start_Time_Ordered_Queue.Remove_Item
                                   ( Latest_Start_Time_Que,
                                     Latest_Start_Time_Ordered_Queue.Position_Of
                                        ( Current_Task, Latest_Start_Time_Que ));
        case Current_Task.Status is

            when PREEMPTED_MANDATORY =>

                Current_Task.Status := EXECUTING_MANDATORY ;

            when PREEMPTED_OPTIONAL =>

                Current_Task.Status := EXECUTING_OPTIONAL ;

            when OTHERS =>
                null ;
        end case ;

        Current_Task.Started_At := Calendar.Clock ;
        V_XTasking.Set_Priority ( Executing_Priority, Current_Task.Task_ID ) ;

    else

        Current_Task := null ;

    end if ;

end Non_Periodic_Completed ;
```

```
separate ( Task_Manager_Package )
    ------------------------------------------------------
    --                                                  --
    --    PROCEDURE  Periodic_Priorities_By_Importance  --
    --                                                  --
    ------------------------------------------------------
    --
    --    This procedure uses an extrenally visible priority que
    -- called Temp_by_Period.  If a task's mandatory part can be done,
    -- then that task is added to the que.  If it cannot, then the
    -- task is assigned the bottom priority.
    --
procedure Periodic_Priorities_By_Importance
                        ( The_Queue  : in Importance_Ordered_Queue.Deque ;
                          Temp_Queue : in out Period_Ordered_Queue.Deque ) is

    Current_Priority : Integer := 0 ;

    Procedure Priorities_By_Imp_Period ( The_TCB  : in Task_Control_Block_Ptr ;
                                         Continue : out Boolean ) is
    begin
        if The_TCB.Periodic_Variables.Mode /= Mandatory then
            The_TCB.Periodic_Variables.Mode := Mandatory ;
            Periodic_Tasks.Store_Variables ( The_TCB.Periodic_Variables,
                                             The_TCB.Task_ID ) ;
        end if;
        if Periodic_Utilization < Required_Utilization then
            Additional_Utilization := ( Float ( The_TCB.Mandatory_Duration )
                                      + Twice_Task_Swap_Delay )
                                    / Float ( The_TCB.Period ) ;

            if ( Periodic_Utilization + Additional_Utilization )
                 <= Required_Utilization then
                Periodic_Utilization := Periodic_Utilization +
                                        Additional_Utilization ;
                Period_Ordered_Queue.Add ( The_TCB,
                                           Temp_Queue,
                                           Period_Queue_Back ) ;
            end if ;
        end if;
        The_TCB.Priority := Bottom_Priority ;
        Continue := True ;
    end Priorities_By_Imp_Period ;
    procedure Traverse_by_Importance is new
       Importance_Ordered_Queue.Iterate ( Process => Priorities_By_Imp_Period ) ;

begin
    Traverse_by_Importance ( The_Queue ) ;
end Periodic_Priorities_By_Importance ;
```

```
separate ( Task_Manager_Package )
    ------------------------------------------------------
    --                                                  --
    --        PROCEDURE  Print_Non_Periodic_Tasks       --
    --                                                  --
    ------------------------------------------------------
    --
    procedure Print_Non_Periodic_Tasks is
    begin
        Text_IO.New_Line ;
        Text_IO.Put ( "  >>>>>>>>>  The Time Now is => " ) ;
        Time_IO.Put ( Calendar.Seconds ( Calendar.Clock ) ) ;
        Text_IO.Put ( "  Periodic Utilization Budget => " ) ;
        Float_IO.Put ( Periodic_Utilization_Budget, Fore => 1, Aft => 2 ) ;
        Text_IO.Put_Line ( " <<<<<<<<<" ) ;
        Text_IO.Put_Line ( "                  Durations        Time              "
                         &"                                                    "
                         &"   Latest" ) ;
        Text_IO.Put_Line ( " Task ID  Mandatory Optional Remaining Start-Time"
                         &"  Deadline  Started-At Importance Status"
                         &" Start-Time" ) ;
        Text_IO.Put_Line ( "--------- --------- -------- --------- ----------"
                         &" ---------- ---------- ---------- ------"
                         &" ----------" ) ;
        Print_Non_Periodic_Tasks ( Tasks_By_ID ) ;
        Text_IO.New_Line ;
    end Print_Non_Periodic_Tasks ;
```

```
separate ( Task_Manager_Package )
    ------------------------------------------------------------
    --                                                        --
    --          PROCEDURE  Print_Periodic_Tasks               --
    --                                                        --
    ------------------------------------------------------------
    --
    procedure Print_Periodic_Tasks is
    begin
        Text_IO.Put_Line ( "                                Durations " ) ;
        Text_IO.Put_Line ( " Task ID   Period   Mandatory   Optional "
                         &"  Kind  Importance Priority   Mode" ) ;
        Text_IO.Put_Line ( "--------- -------- ----------- ----------"
                         &"------ ---------- -------- ---------" ) ;
        Print_Tasks_By_Period ( Tasks_By_Period ) ;
        Text_IO.New_Line ;
        Text_IO.Put ( "Periodic Utilization  => " ) ;
        Float_IO.Put ( Periodic_Utilization ) ;
        Text_IO.New_Line ;
        Text_IO.Put ( "Mandatory Utilization => " ) ;
        Float_IO.Put ( Mandatory_Utilization ) ;
        Text_IO.New_Line ;
        Text_IO.Put ( "Optional Utilization  => " ) ;
        Float_IO.Put ( Optional_Utilization ) ;
        Text_IO.New_Line ;
        Text_IO.Put ( "Required Utilization  => " ) ;
        Float_IO.Put ( Required_Utilization ) ;
        Text_IO.New_Line ;
        Text_IO.Put ( "Periodic Utilization Budget => " ) ;
        Float_IO.Put ( Periodic_Utilization_Budget ) ;
        Text_IO.New_Line ;
        Text_IO.Put ( "Current Periodic Condition  => " ) ;
        Text_IO.Put ( " " & Periodic_Condition_Type'Image(Periodic_Condition)) ;
        Text_IO.New_Line ( 2 ) ;
    end Print_Periodic_Tasks ;
```

```
separate ( Task_Manager_Package )
    ------------------------------------------------------------
    --                                                        --
    --          PROCEDURE  Print_Test_Times                   --
    --                                                        --
    ------------------------------------------------------------
    --
    procedure Print_Test_Times ( Test_Number : in integer ) is
    begin
        if Test_Number = 1 or Test_Number = 5 then
            Text_IO.Put_Line (" ----------  Add Times  ---------- " ) ;
            Text_IO.Put_Line ("#Per   #Non_P   MIN      MAX      SUM " ) ;
            for I in 1..Max_Periodic loop
                for j in 1..Max_Non_Periodic loop
                    Text_IO.Put ( Integer'Image(i) & "   " ) ;
                    Text_IO.Put ( Integer'Image(j) & "   " ) ;
                    Float_IO.Put(Test_Adds(MIN,i,j),Fore => 1,Aft => 4,Exp => 0);
                    Text_IO.Put ( "   " ) ;
                    Float_IO.Put(Test_Adds(MAX,i,j),Fore => 1,Aft => 4,Exp => 0);
                    Text_IO.Put ( "   " ) ;
                    Float_IO.Put(Test_Adds(SUM,i,j),Fore => 1,Aft => 4,Exp => 0);
                    Text_IO.New_Line ;
                end loop ;
            end loop ;
        end if ;
        if Test_Number = 2 or Test_Number = 5 then
            Text_IO.New_Line ;
            Text_IO.Put_Line (" ----------  Modify Times  ---------- " ) ;
            Text_IO.Put_Line ("#Per   #Non_P   MIN      MAX      SUM " ) ;
            for I in 1..Max_Periodic loop
                for j in 1..Max_Non_Periodic loop
                    Text_IO.Put ( Integer'Image(i) & "   " ) ;
                    Text_IO.Put ( Integer'Image(j) & "   " ) ;
                    Float_IO.Put(Test_Mods(MIN,i,j),Fore => 1,Aft => 4,Exp => 0);
                    Text_IO.Put ( "   " ) ;
                    Float_IO.Put(Test_Mods(MAX,i,j),Fore => 1,Aft => 4,Exp => 0);
                    Text_IO.Put ( "   " ) ;
                    Float_IO.Put(Test_Mods(SUM,i,j),Fore => 1,Aft => 4,Exp => 0);
                    Text_IO.New_Line ;
                end loop ;
            end loop ;
        end if ;
        if Test_Number = 3 or Test_Number = 5 then
            Text_IO.New_Line ;
            Text_IO.Put_Line (" ----------  Remove Times  ---------- " ) ;
            for I in 1..Max_Periodic loop
                for j in 1..Max_Non_Periodic loop
                    Text_IO.Put ( Integer'Image(i) & "   " ) ;
                    Text_IO.Put ( Integer'Image(j) & "   " ) ;
                    Float_IO.Put(Test_Rems(MIN,i,j),Fore => 1,Aft => 4,Exp => 0);
                    Text_IO.Put ( "   " ) ;
                    Float_IO.Put(Test_Rems(MAX,i,j),Fore => 1,Aft => 4,Exp => 0);
                    Text_IO.Put ( "   " ) ;
                    Float_IO.Put(Test_Rems(SUM,i,j),Fore => 1,Aft => 4,Exp => 0);
                    Text_IO.New_Line ;
                end loop ;
            end loop ;
        end if ;
        if Test_Number = 4 or Test_Number = 5 then
            Text_IO.New_Line ;
            Text_IO.Put_Line (" ----------  Change PU Times  ---------- " ) ;
            Text_IO.Put_Line ("#Tasks  MIN      MAX      SUM " ) ;
            for I in 1..Max_Periodic loop
                for j in 1..Max_Non_Periodic loop
                    Text_IO.Put ( Integer'Image(i) & "   " ) ;
                    Text_IO.Put ( Integer'Image(j) & "   " ) ;
                    Float_IO.Put(Test_PUs(MIN,i,j),Fore => 1,Aft => 4,Exp => 0);
                    Text_IO.Put ( "   " ) ;
                    Float_IO.Put(Test_PUs(MAX,i,j),Fore => 1,Aft => 4,Exp => 0);
                    Text_IO.Put ( "   " ) ;
                    Float_IO.Put(Test_PUs(SUM,i,j),Fore => 1,Aft => 4,Exp => 0);
                    Text_IO.New_Line ;
```

```
            end loop ;
        end loop ;
    end if ;
    if Test_Number = 6 then
        Text_IO.New_Line ;
        Text_IO.Put_Line (" ----------  Task Completed Times  ---------- " ) ;
        Text_IO.Put_Line ("#Tasks  MIN       MAX       SUM " ) ;
        for I in 1..Max_Periodic loop
            for j in 1..Max_Non_Periodic loop
                Text_IO.Put ( Integer'Image(i) & "    " ) ;
                Text_IO.Put ( Integer'Image(j) & "    " ) ;
                Float_IO.Put(Test_Comps(MIN,i,j),Fore => 1,Aft => 4,Exp => 0);
                Text_IO.Put ( "    " ) ;
                Float_IO.Put(Test_Comps(MAX,i,j),Fore => 1,Aft => 4,Exp => 0);
                Text_IO.Put ( "    " ) ;
                Float_IO.Put(Test_Comps(SUM,i,j),Fore => 1,Aft => 4,Exp => 0);
                Text_IO.New_Line ;
            end loop ;
        end loop ;
    end if ;
    if Test_Number = 7 then
        Print_Periodic_Tasks ;
    elsif Test_Number = 8 then
        Print_Non_Periodic_Tasks ;
    end if;
end Print_Test_Times ;
```

```
separate ( Task_Manager_Package )
    ------------------------------------------------------------
    --                                                        --
    --              PROCEDURE  Record_Times                   --
    --                                                        --
    ------------------------------------------------------------
    --
    procedure Record_Times ( Start_Time : in Calendar.Time ;
                             Stop_Time  : in Calendar.Time ;
                             Num_NP_Ts  : in integer ;
                             Num_P_Ts   : in integer ;
                             The_Array  : in out Test_Values_Array ) is
        Elapsed_Time : Float ;
    begin
        Elapsed_Time := Float ( Stop_Time - Start_Time ) ;

        if Elapsed_Time < The_Array ( MIN, Num_P_Ts, Num_NP_Ts ) then
            The_Array ( MIN, Num_P_Ts, Num_NP_Ts ) := Elapsed_Time ;
        end if;
        if Elapsed_Time > The_Array ( MAX, Num_P_Ts, Num_NP_Ts ) then
            The_Array ( MAX, Num_P_Ts, Num_NP_Ts) := Elapsed_Time ;
        end if;

        The_Array ( SUM, Num_P_Ts, Num_NP_Ts ) :=
                    The_Array ( SUM, Num_P_Ts, Num_NP_Ts ) + Elapsed_Time ;
    end Record_Times ;
```

```
separate ( Task_Manager_Package )
    ------------------------------------------------------
    --                                                  --
    --                PROCEDURE Schedule                --
    --                                                  --
    ------------------------------------------------------

procedure Schedule ( The_TCB : in out Task_Control_Block_Ptr ) is

begin

    The_TCB.Status := READY ;
    case The_TCB.Task_Kind is

        when ANY_TIME =>
            --
            --    This is the first place that a non-periodic task is added
            -- to the queue structures.  This is assumed to be called after
            -- the feasibility of a task is determined.  Non-feasible non-
            -- periodic tasks are not allowed into the system.
            --
            Any_Time_Tasks.Store_Variables ( The_TCB.Any_Time_Variables,
                                             The_TCB.Task_ID ) ;
        when PERIODIC =>
            Assign_Periodic_Priorities ;
            Periodic_Tasks.Store_Variables ( The_TCB.Periodic_Variables,
                                             The_TCB.Task_ID ) ;

        when SINGULAR =>
            Singular_Tasks.Store_Variables ( The_TCB.Singular_Variables,
                                             The_TCB.Task_ID ) ;
    end case ;

    Start_Time_Ordered_Queue.Add ( The_TCB, Ready_Que, Ready_Queue_Back ) ;

end Schedule ;
```

```
separate ( Task_Manager_Package )
    ------------------------------------------------------
    --                                                  --
    --         PROCEDURE  Some_Periodics_Optional       --
    --                                                  --
    ------------------------------------------------------
    --
    --    This procedure is used to adjust the execution mode of those
    -- tasks that can execute their optional parts as well.  The
    -- Periodic_Utilization is addjusted as the task's optional parts are
    -- added.  Note that it is possible for a task of a lower importance
    -- with a shorter optional duration to be scheduled to do its
    -- optional part.
    --
procedure Some_Periodics_Optional
                        ( The_Queue : in Importance_Ordered_Queue.Deque ) is

    Procedure Adjust_Periodic_Modes ( The_TCB  : in Task_Control_Block_Ptr ;
                                      Continue : out Boolean ) is
    begin
        if Periodic_Utilization < Required_Utilization then
            Additional_Utilization := ( Float ( The_TCB.Optional_Duration )
                                        + Twice_Task_Swap_Delay )
                                      / Float ( The_TCB.Period ) ;
            if ( Periodic_Utilization + Additional_Utilization )
                 <= Required_Utilization then
                Periodic_Utilization := Periodic_Utilization +
                                        Additional_Utilization ;
                The_TCB.Periodic_Variables.Mode := Optional ;
                Periodic_Tasks.Store_Variables ( The_TCB.Periodic_Variables,
                                                 The_TCB.Task_ID ) ;
            end if ;
        end if;
        Continue := True ;
    end Adjust_Periodic_Modes ;
    procedure Traverse_Periodics_Optional is
              new Importance_Ordered_Queue.Iterate
                                        ( Process => Adjust_Periodic_Modes ) ;
begin
    Traverse_Periodics_Optional ( The_Queue ) ;
end Some_Periodics_Optional ;
```

```
separate ( Task_Manager_Package )
    ------------------------------------------------------
    --                                                  --
    --                PROCEDURE  Un_Schedule            --
    --                                                  --
    ------------------------------------------------------
procedure Un_Schedule ( The_TCB : in Task_Control_Block_Ptr ) is
begin
    case The_TCB.Task_Kind is
        --
        --    A periodic task exists in the TASKS_BY_PERIOD, READY, and the
        -- PERIODIC_IMPORTANCE queues.  To unschedule a periodic task
        -- involves;    1) removing it from the queues
        --              2) adjusting the values used to assign priorities
        --              3) rescheduling the remaining periodics
        --              4) rescheduling the non-periodic tasks
        --
        when PERIODIC =>
            Period_Ordered_Queue.Remove_Item
                      ( Tasks_By_Period,
                        Period_Ordered_Queue.Position_Of ( The_TCB,
                                                           Tasks_By_Period )) ;
            Start_Time_Ordered_Queue.Remove_Item
                      ( Ready_Que,
                        Start_Time_Ordered_Queue.Position_Of ( The_TCB,
                                                               Ready_Que )) ;
            Importance_Ordered_Queue.Remove_Item
                      ( Periodic_Importance_Que,
                        Importance_Ordered_Queue.Position_Of
                                    (The_TCB, Periodic_Importance_Que)) ;
            Number_of_Periodic_Tasks := Number_of_Periodic_Tasks - 1 ;
            --
            -- The goal here is to determine the size of each priority
            -- bin.  That means that given some number of priorities
            -- and a set of tasks whose cardinality is larger than the
            -- number of priorities and each with an unrestrained period,
            -- what should the range of periods for each available
            -- priority be?  This is only calculated when there are more
            -- tasks than priorities to save a little on overhead.
            --
            if Number_of_Periodic_Tasks > Number_of_Periodic_Priorities then
                Periodic_Bin_Size := Integer ( Number_of_Periodic_Tasks /
                                    Number_of_Periodic_Priorities ) ;
            end if ;

            Mandatory_Utilization := Mandatory_Utilization -
                        ( Float ( The_TCB.Mandatory_Duration )
                                + Twice_Task_Swap_Delay )
                        / Float ( The_TCB.Period ) ;
            Optional_Utilization := Optional_Utilization -
                        ( Float ( The_TCB.Mandatory_Duration )
                                + Float ( The_TCB.Optional_Duration )
                                + Twice_Task_Swap_Delay )
                        / Float ( The_TCB.Period ) ;
            case The_TCB.Periodic_Variables.Mode is
                when MANDATORY =>
                    Periodic_Utilization := Periodic_Utilization -
                        ( Float ( The_TCB.Mandatory_Duration )
                                + Twice_Task_Swap_Delay )
                        / Float ( The_TCB.Period ) ;
                when OPTIONAL =>
                    Periodic_Utilization := Periodic_Utilization -
                        ( Float ( The_TCB.Mandatory_Duration )
                                + Float ( The_TCB.Optional_Duration )
                                + Twice_Task_Swap_Delay )
                        / Float ( The_TCB.Period ) ;
            end case ;

            The_TCB.Status := COMPLETED ;
            The_TCB.Next :=  Free_Task_List ( The_TCB.Task_Kind ) ;
            Free_Task_List ( The_TCB.Task_Kind ) := The_TCB ;

        when ANY_TIME | SINGULAR =>
            --
            --    The first thing to do is to insure that the task knows it
            -- is not to continue.  This will ensure that it ends when it
            -- reaches a termination point.
            --
            case The_TCB.Task_Kind is
                when ANY_TIME =>
                    The_TCB.Any_Time_Variables.Continue := FALSE ;
                    Any_Time_Tasks.Store_Variables ( The_TCB.Any_Time_Variables,
                                                     The_TCB.Task_ID ) ;
                when SINGULAR =>
                    The_TCB.Singular_Variables.Continue := FALSE ;
                    Singular_Tasks.Store_Variables ( The_TCB.Singular_Variables,
                                                     The_TCB.Task_ID ) ;
                 when OTHERS =>
                     null ;
            end case ;
            --
            --    Adjust the number of active non-periodic tasks
            --
            Num_of_NP_Tasks := Num_of_NP_Tasks - 1 ;

            case The_TCB.Status is
                --
                --    If the task is in the Ready state, then that means it
                -- has not yet reached its start time and is only in the
                -- start time que.
                --
                when READY =>
                    Start_Time_Ordered_Queue.Remove_Item
                              ( Ready_Que,
                                Start_Time_Ordered_Queue.Position_Of
                                                    ( The_TCB, Ready_Que ));
                    The_TCB.Status := COMPLETED ;
                    The_TCB.Next :=  Free_Task_List ( The_TCB.Task_Kind ) ;
                    Free_Task_List ( The_TCB.Task_Kind ) := The_TCB ;
                when  EXECUTING_MANDATORY | EXECUTING_OPTIONAL =>
                    --
                    --    The task is currently executing so we need to set its
                    -- priority to a lower value and change its status.  When
                    -- it completes, i.e. reaches a safe place to terminate,
                    -- then it will be placed back on the free list.
                    --
                    V_XTasking.Set_Priority ( Discarded_Priority,
                                              The_TCB.Task_ID ) ;
                    The_TCB.Status := DISCARDED ;
                when PREEMPTED_MANDATORY | PREEMPTED_OPTIONAL =>
                    --
                    --    This task has been started and then preempted so it
                    -- is in both the latest start time and the the deadline
                    -- queues.  So it needs to be removed from them.
                    --
                    V_XTasking.Set_Priority ( Discarded_Priority,
                                              The_TCB.Task_ID ) ;
                    The_TCB.Status := DISCARDED ;
                    Deadline_Ordered_Queue.Remove_Item
                                    ( Deadline_Que,
                                      Deadline_Ordered_Queue.Position_Of
                                                ( The_TCB, Deadline_Que ));
                    Latest_Start_Time_Ordered_Queue.Remove_Item
                                ( Latest_Start_Time_Que,
                                  Latest_Start_Time_Ordered_Queue.Position_Of
                                            ( The_TCB, Latest_Start_Time_Que ));
                when OTHERS =>  null ;

            end case ;

    end case ;

end Un_Schedule ;
```

```
;;;
;;;   Example Reasoning Process Rules
;;;

(defrule RESPOND_TO_EVENT_1
        "Event 1 generates an anytime task"
          ?Event <- (Event_Message 1 ?time)
          =>
          (retract ?Event)
          (bind ?start_time (+ ?time 5.0))
          (bind ?deadline   (+ ?time 10.0))
          (bind ?kind 5)
          (assert (ANY_TIME =(add_task TRUE
                                                ANY_TIME
                                    ?kind
                                    0.0
                                    ?deadline
                                    ?start_time
                                    3)))
)

(defrule RESPOND_TO_EVENT_2
        "Event 2 generates an periodic task"
          ?Event <- (Event_Message 2 ?time)
          =>
          (retract ?Event)
          (bind ?start_time (+ ?time 5.0))
          (bind ?deadline   (+ ?time 10.0))
          (bind ?period (* 0.00000001 (random)))
          (bind ?kind 5)
          (bind ?importance 4)
          (assert (PERIODIC =(add_task TRUE
                                                PERIODIC
                                    ?kind
                                    ?period
                                    ?deadline
                                    ?start_time
                                    ?importance)))
)

(defrule RESPOND_TO_EVENT_3
        "Event 3 generates an singular task"
          ?Event <- (Event_Message 3 ?time)
          =>
          (retract ?Event)
          (bind ?start_time (+ ?time 5.0))
          (bind ?deadline   (+ ?time 10.0))
          (bind ?period 0.0 )
          (bind ?kind 5)
          (bind ?importance 4)
          (assert (SINGULAR =(add_task TRUE
                                                SINGULAR
                                    ?kind
                                    ?period
                                    ?deadline
                                    ?start_time
                                    ?importance)))
)

(defrule RESPOND_TO_EVENT_4
        "Event 4 terminates an any-time task"
          ?Event <- (Event_Message 4 ?time)
        ?Task  <- (ANY_TIME ?task_id)
          =>
          (retract ?Event)
          (retract ?Task)
        (remove_task ?task_id)
)

(defrule RESPOND_TO_EVENT_5
        "Event 5 terminates a periodic task"
          ?Event <- (Event_Message 5 ?time)
        ?Task  <- (PERIODIC ?task_id)
          =>
          (retract ?Event)
          (retract ?Task)
        (remove_task ?task_id)
)

(defrule RESPOND_TO_EVENT_6
        "Event 6 terminates a singular task"
          ?Event <- (Event_Message 6 ?time)
        ?Task  <- (SINGULAR ?task_id)
          =>
          (retract ?Event)
          (retract ?Task)
        (remove_task ?task_id)
)

(defrule RESPOND_TO_EVENT_7
        "Event 7 terminates a singular task"
          ?Event <- (Event_Message 7 ?time)
          =>
          (retract ?Event)
)

(defrule RESPOND_TO_EVENT_8
        "Event 8 terminates a singular task"
          ?Event <- (Event_Message 8 ?time)
          =>
          (retract ?Event)
)

(defrule RESPOND_TO_EVENT_9
        "Event 9 terminates a singular task"
          ?Event <- (Event_Message 9 ?time)
          =>
          (retract ?Event)
)

(defrule REMOVE_INFEASIBLE_TASK
          "Removes the fact assoicated with an infeasible task"
          ?message    <- (Infeasible_Task ?task_id)
        ?infeasible <- (?task_type ?task_id)
        =>
        (retract ?infeasible)
        (retract ?message)
)

(defrule PRINT_STATUS
          ?status <- (Task_Manager_Status ?pu ?mu ?ou ?ru ?cond ?numPT ?NumNPT)
          =>
          (retract ?status)
)
```

Bibliography

[AdaLRM, 1983]. Department of Defense. *Reference Manual For The Ada Programming Language.* ANSI/MIL-STD-1815 A. Washington: Ada Joint Program Office, Office of the Under Secretary of Defense Research and Engineering, January 1983.

[Aldern, 1990]. Aldern, Thomas D., and others. *Phase I Final Report of the Pilot's Associate Program, Interim Report for Period November 1989–December 1990.* Wright Research and Development Center Technical Report WRDC-TR-90-7007. Marietta, GA: Lockheed Aeronautical Systems Company, December 1990. (Limited Distribution – Distribution authorized to DoD and DoD contractors only).

[Aldern, 1991]. Aldern, Thomas D., and others. *Phase 2 Interim Report of the Pilot's Associate Program, Interim Report for Period November 1989–December 1990.* Wright Laboratories Technical Report WL-TR-91-7007. Marietta, GA: Lockheed Aeronautical Systems Company, October 1991. (Limited Distribution - Distribution authorized to DoD and DoD contractors only).

[Allen, 1990]. Allen, Arnold O. *Probability, Statistics, and Queueing Theory with Computer Science Applications* (Second Edition). San Diego, CA: Academic Press Inc., 1990.

[ARTIE, 1989]. Boeing Military Airplanes. *Ada Real-Time Inferene Engine (ARTIE) User's Guide.* Version 1.1. Witchita, K : Avionics Technology, March 1989.

[Banks, 1991]. Banks, Sheila M., Lizza, Carl S., and Whelan, Michael A. "Pilot's Associate: Evolution of a Functional Prototype," *AGARD Conference Proceedings Machine Intelligence for Aerospace Electronic Systems.* pp. 16-1 thru 16-12. 7 Rue Ancelle 92200, Neuilly sur Seine, France: Advisory Group for Aerospace Research & Development, May 1991.

[Baruah, 1991]. Baruah, S., and others. "On-line Scheduling in the Presence of Overload," *Proceedings of the 32nd Annual Symposium On Foundations of Computer Science.* pp. 100–110. Los Alamitos CA: IEEE, IEEE Computer Society Press, 1991.

[Bihari, 1989]. Bihari, Thomas E., Walliser, Thomas M., and Patterson, Mark R. "Controlling the Adaptive Suspension Vehicle," *IEEE Computer, 22, 6* : 59-95 (June 1989).

[Booch, 1983]. Booch, Grady. *Software Engineering with Ada.* Menlo Park, CA: The Benjamin/Cummings Publishing Company, Inc., 1983.

[Booch, 1986]. Booch, Grady. *Software Components with Ada, Structures, Tools, and Subsystems.* Menlo Park, CA: The Benjamin/Cummings Publishing Company, Inc., 1986.

[Borger, 1989]. Borger, Mark W., Klein, Mark H., and Veltre, Robert A. *Real-Time Software Engineering in Ada: Observations and Guidlines.* Technical Report CMU/SEI-89-TR-22. Carnegie Mellon University, Pittsburgh, PA: Software Engineering Institute, September 1989.

[Brassard, 1988]. Brassard, Gilles and Bratley, Paul. *Algorithmics Theory & Practice*, Englewood Cliffs, NJ: Prentice Hall, 1988.

[Broger, 1989]. Broger, Mark W. and Rajkumar, Ragunathan. *Implementing Priority Inheritance Algorithms in an Ada Runtime System.* Technical Report CMU/SEI-89-TR-15. Carnegie Mellon University, Pittsburgh, PA: Software Engineering Institute, April 1989.

[ClassicAda, 1989]. Software Productivity Solutions.*Classic-Ada User's Manual.* 1989.

[CLIPS-Ada, 1991]. NASA. *CLIPS/Ada Advanced Programming Guide.* Version 4.4, Revision 1. Johnson Space Center: Information Systems Directorate, Software Technology Branch, October 1991.

[CLIPSRefMan, 1991a]. NASA. *CLIPS Reference Manual, Volume I, Basic Programming Guide*. CLIPS Version 5.1. Johnson Space Center: Information Systems Directorate, Software Technology Branch, September 1991.

[CLIPSRefMan, 1991b]. NASA. *CLIPS Reference Manual, Volume II, Advanced Programming Guide*. Version 5.1. Johnson Space Center: Information Systems Directorate, Software Technology Branch, September 1991.

[CLIPSRefMan, 1991c]. NASA. *CLIPS Reference Manual, Volume III, Utilities and Interfaces Guide*. Version 5.1. Johnson Space Center: Information Systems Directorate, Software Technology Branch, September 1991.

[CLIPSUG, 1991]. NASA. *CLIPS User's Guide Volume I - Rules*. Version 5.1. Johnson Space Center: Information Systems Directorate, Software Technology Branch, September 1991.

[Coffman, 1976]. Coffman, Edward G. *Computer and Job–Shop Scheduling Theory*. New York, NY: John Wiley & Sons, Inc, 1976.

[Cohen, 1986]. Cohen, Norman H. *Ada as a Second Language*. New York, NY: McGraw-Hill Book Company, 1986.

[Cormen, 1990]. Cormen, Thomas H., Leiserson, Charles E., and Rivest, Ronald L. *Introduction to Algorithms*. Cambridge, MA: The MIT Press, 1990.

[Dechter, 1991]. Dechter, Rina, Miri, Itay, and Pearl, Judea "Temporal constraint networks," *Artificial Intelligence, 49, 1–3* : 61–95 (May 1991).

[Dodhiawala, 1988]. Dodhiawala, Rajendra and Sridharan, N.S. *Real-Time Impact Report (RT-1 Impact Analysis)*. MCAIR SDRL 10-1. Santa Clara, CA: FMC Corporation, Central Engineering Laboratories, September 1988.

[Dodhiawala, 1989]. Dodhiawala, Rajendra, Sridharan, N.S., Raulefs, Peter, and Pickering, Cynthia "Real-Time AI Systems: A Definition and An Architecture," *Eleventh International Joint Conference on Artificial Intelligence*. pp. 256 -261. San Mateo, CA: The International Joint Conferences on Artificial Intelligence, Inc., Morgan Kaufmann Publishers, Inc., August 1989.

[Fanning, 1990]. Fanning, Franklin Jesse. *An Evaluation of an Ada Implementation of the Rete Algorithm for Embedded Flight Processors*. MS thesis, AFIT/GE/ENG/90D-70. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1990.

[Klahr, 1986]. Kalhr, Phillp and Waterman, Donald A. *Expert Systems Techniques, Tools, and Applications*. Reading, MA: Addison-Wesley Publishing Company, 1986.

[Klein, 1990]. Klein, Mark H. and Ralya, Thomas. *An Analysis of Input/Output Paradigms for Real-Time Systems*. Technical Report CMU/SEI-90-TR-19. Carnegie Mellon University, Pittsburgh, PA: Software Engineering Institute, July 1990.

[Lambert, 1990]. Lambert, R.E., and others. *Phase I Final Report of the Pilot's Associate Program, Final Report for Period Febuary 1986–December 1990*. Wright Laboratories Technical Report WL-TR-91-7006. St. Louis, MO: McDonnell Aircraft Company, December 1990. (Limited Distribution - Distribution authorized to DoD and DoD contractors only).

[Lambert, 1991]. Lambert, R.E., and others. *Technical Operating Report - System Design Document, Final Report for Period August 1988–December 1990*. Wright Laboratory Technical Report WL-TR-91-7005. St. Louis, MO: McDonnell Aircraft Company, September 1991. (Limited Distribution - Distribution authorized to DoD and DoD contractors only).

[Lamont, 1991]. Lamont, Gary B. *Real–Time Scheduling of Periodic and Aperiodic Tasks*. February 1991, CSCE686 Class Notes, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, February 1991.

[Liu, 1973]. Liu, C. L. and Layland, James W. "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *Journal of the Association for Computing Machinery, 20, 1* : 46-61 (January 1973).

[Liu, 1991]. Liu, Jane W.S., and others. "Algorithms for Scheduling Imprecise Computations," *IEEE Computer, 24, 5* : 58–68 (May 1991).

[Lizza, 1989]. Lizza, Carl S. "Pilot's Associate: A Perspective Demonstration 2 ," *Proceeding of Computers in Aerospace Conference.* . AIAA, 1989.

[Locke, 1992]. Locke, C. Douglass and Vogel, David R. *Ada Real-Time Programming: A Seminar*. March 1992, IBM Federal Sector Division, Owego, NY 13827.

[Nii, 1989]. Nii, H. Penny. *The Handbook of Artificial Intelligence*, Volume IV. Reading, MA: Addison-Wesley Publishing Company, 1989.

[O'Reilly, 1988]. O'Reilly, Cindy A. and Cromarty, Andrew S. "'Fast' is not 'Real-Time': Designing Effective Real-Time AI Systems," *Applications of Artificial Intelligence II*. pp. 249–257. SPIE, SPIE, 1988.

[Payton, 1991]. Payton, David W. and Bihari, Thomas E. "Intelligent Real-Time Control of Robotic Vehicles," *Communications of the ACM, 34, 8* : 48–63 (August 1991).

[Real-Time, 1984]. U.S. Army Communications-Electronics Command. *Real-Time Ada Workbook*. Fort Monmouth, NJ: Center For Tactical Computer Systems, July 1984.

[Sawyer, 1990]. Sawyer, George Allen. *Extraction and Measurement of Multi-Level Parallelism in Production Systems*. MS thesis, AFIT/GCE/ENG/90D-04. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH. December 1990.

[Scoy, 1989]. Scoy, Roger Van, Bamberger, Judy, and Firth, Robert. *An Overview of DARK,* Ada-Letters (November/December 1989).

[Sha, 1989]. Sha, Lui and Goodenough, John B. *Real–Time Scheduling Theory and Ada.* Technical Report CMU/SEI-89-TR-14. Carnegie Mellon University, Pittsburgh, PA: Software Engineering Institute, April 1989.

[Sha, 1991]. Sha, Lui, Klein, Mark H., and Goodenough, John B. *Rate Monotonic Analysis for Real-Time Systems*. Technical Report CMU/SEI-91-TR-6. Carnegie Mellon University, Pittsburgh, PA: Software Engineering Institute, March 1991.

[Shamsudin, 1991]. Shamsudin, Annie Z. and Dillion, T.S. *NetManager: A Real--Time Expert System for Network Traffic Management*. Technical Report 15/91. Department of Computer Science and Computer Engineering, La Trobe University, Bundoora, Victoria, Australia 3083: La Trobe University, December 1991.

[Simpson, 1988]. Simpson, Robert L. "DoD Applications of Artificial Intelligence: Successes and Prospects," *Applications of Artificial Intelligence VI.* . SPIE, SPIE, 1988.

[Smith, 1990]. Smith, David M. and Broadwell, Martin M. *Pilot's Associate System Knowledge Base Document, Volume 1: Tactics Planner Subsystem, CDRL Sequence No. 32,* . Contract F33615-85-C-3804. Marietta, GA: Lockheed Aeronautical Systems Company, October 1990. (Distribution Limited to DoD and DoD contractors only).

[Sprunt, 1989]. Sprunt, Brinkley, Sha, Lui, and Lehoczky, John. *Scheduling Sporadic and Aperiodic Events in a Hard Real-Time System*. Technical Report CMU/SEI-89-TR-11. Carnegie Mellon University, Pittsburgh, PA: Software Engineering Institute, April 1989.

[Sprunt, 1990]. Sprunt, Brinkley and Sha, Lui. *Implementing Sporadic Servers in Ada*. Technical Report CMU/SEI-90-TR-6. Carnegie Mellon University, Pittsburgh, PA: Software Engineering Institute, May 1990.

[Stankovic, 1988]. Stankovic, John A. and Ramamritham, Krithi, *Tutorial Hard Real-Time Systems*, IEEE Catolog Number EH0276-6, Computer Society Press Order Number 81, February 1988.

[Stockman, 1988]. Stockman, Steven P. "ABLE: An Ada--Based Blackboard System," *Proceedings of AIDA-88, Fourth International Conference on Artificial Intelligence and Ada*. George Mason University, George Mason University, 1988.

[Tindell, 1992]. Tindell, Ken, Burns, Alan, and Wellings, Andy, *Allocating Hard Real Time Tasks (An NP–Hard Problem Made Easy)*, e-mail via ftp, 1992, Real Time Systems Research Group, Department of Computer Science, University of York, England.

[VERDIX, 1990]. VERDIX Corporation. *VADS Veridx Ada Development System Version 6.0*. Sun-4 Sun OS. August 1990.

[Whelan, 1990]. Whelan, Michael A. and Rouse, Doug Pilot's Associate: Approaching Maturity. In *Seventh Annual Workshop on Command and Control Decision Aiding*. Air Force Institute of Technology, Valusek, J. R. and Duffy, LorRaine, Ch. 3, Air Force Institute of Technology/ENS Wright-Patterson AFB, Ohio 45433-6583, Distribution Limited to DoD and DoD contractors only, April 1990.

[Wilber, 1989]. Wilber, George F. "Intelligent Real-Tim Embedded Systems," *Proceedings of AIDA-89, Fifth International Conference on Artificial Intelligence and Ada*. pp. 74–82. Washington D.C.: Department of Computer Science, George Mason University and The Institute for Defense Analyses, George Mason University, November 1989.

[Wilensky, 1983]. Wilensky, Robert *Planning and Understanding A Computational Approach to Human Reasoning*. Reading, MA: Addison-Wesley Publishing Company, 1983.

[Wood, 1989]. Wood, William G. *Temporal Logic Case Study*. Technical Report CMU/SEI-89-TR-24. Carnegie Mellon University, Pittsburgh, PA: Software Engineering Institute, August 1989.

Vita

Captain Michael A. Whelan was born to Dr. William J. Whelan and Barbara A. Whelan on July 25, 1958. He enlisted in the United States Air Force on February 22, 1977 as an Aircraft Armament Systems Specialist. Captain Whelan served enlisted tours at Lowry AFB, Colorado; Cannon AFB, New Mexico; RAF Upper Heyford, United Kingdom; and Moody AFB, Georgia. His enlisted positions included Weapons Load Crew Chief, Shift Supervisor, and Weapons Controller in Maintenance Control. Captain Whelan obtained the enlisted rank of Technical Sergeant before entering into the Airman Education and Commissioning program in May of 1985. After graduating with honors from New Mexico State University, Captain Whelan attended Officer Training School and was commissioned as a Second Lieutenant on April 13, 1988. His first officer assignment was to the Cockpit Integration Directorate, Wright Laboratories, Wright-Patterson AFB, Ohio, where he served as the project engineer for the Pilot's Associate program from June 1988 until June 1990. Captain Whelan was then assigned as the Chief Engineer for Multi-Role Cockpit program. In May 1991, Captain Whelan was accepted into the graduate computer engineering program at the Air Force Institute of Technology. Captain Whelan is slated to fill a Command Directed Educational Requirement (CDERs) slot at Wright-Patterson AFB after graduating.

Permanent Address: 6370 Copper Pheasant Drive
Dayton, Ohio 45424-4100
(513) 237-9502

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1992	Master's Thesis

4. TITLE AND SUBTITLE

AN INTELLIGENT REAL-TIME SYSTEM ARCHITECTURE IMPLEMENTED IN ADA

5. FUNDING NUMBERS

6. AUTHOR(S)

Michael A. Whelan, Capt, USAF

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Air Force Institute of Technology, WPAFB OH 45433-6583

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT/GCE/ENG/92D-12

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Major Carl Lizza
WL/FIPA
WPAFB, OH 45433-6553

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

Conventional real-time systems are fully deterministic allowing for off-line, optimal, task scheduling under all circumstances. Real-time intelligent systems add non-deterministic task execution times and non-deterministic task sets for scheduling purposes. Non-deterministic task sets force intelligent real-time systems to trade-off execution time with solution quality during run-time and perform dynamic task scheduling. Four basic design considerations addressing those tradeoffs have been identified: control reasoning, focus of attention, parallelism, and algorithm efficacy. Non-real-time intelligent systems contain an environment sensor, a model of the environment, a reasoning process, and a large collection of procedural processes. Real-time intelligent systems add to these a model of the real-time system's behavior, and a real-time task scheduler. In addition, the reasoning process is augmented with a metaplanner to reason about timing issues using the system's behavioral model. Additionally, real-time deadlines force the inclusion of pluralistic solution methods in the intelligent system to allow multiple responses ranging from reactive to fully reasoned and calculated. This research presents an architecture capable of meeting real-time performance goals with on-line scheduling of tasks.

14. SUBJECT TERMS

Artificial Intelligence, Real-Time, Ada, On-line Scheduling, Knowledge-based Systems, Expert Systems

15. NUMBER OF PAGES

180

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL